Appendix A

```
#-----------------------------------------------------------------------
#
# File: trace.py
#
#      Implements a ray tracing technique and related interpolation and
#      location procedures on 3D tetrahedral meshes. This class is
#      closely related to the Geometry class. This example is based on
#      a tet mesh, but the same functions could be provided for other
#      mesh types.
#
#      Three major example functions are provided:
#        1) cellNumb - Given a set of (x,y,z) coordinates, finds the cell
#           which contains those coordinates. This could be used for
#           example to assign a CT voxel with known coordinates
#           to a cell on the tet mesh.
#        2) fieldVal - Given a set of (x,y,z) coordinates and field values
```

```
#           at the mesh cell vertexes, determines the LDFEM field value
#           at those coordinates. This could be used in a post
#           processing utility to provide exact values at specified
#           points in the problem.
#        3) trackData - Given start and end (x,y,z) coordinates of a
#           track, makes a list of the mesh cells and the path lengths in
#           each cell that the the track passes through. This could be
#           useful in a analytic ray trace algorithm to calculate
#           total optical path length along a ray when cells have
#           different materials assigned to them. The ray starting
#           point must lie inside the mesh, but the ending point can
#           be external. This is useful when modeling external point
#           sources for example.
#
#     Searches through the mesh are expedited in this class by
#     employing a line search based algorithm that reduces the number
#     of cells to be searched. This algorithm effectively bins the
#     mesh cells. Other binning-like algorithms could be envisioned.
#     A production algorithm would include numerous tests and error
#     traps that are not included herein for the sake of simplicity.
#
#
# Author: John McGhee, Radion Technologies
#
# Date  : 09 March 2004

#-----------------------------------------------------------------------
#
# $Id: trace.py,v 1.3 2004/03/10 17:17:26 mcghee Exp $
#
#-----------------------------------------------------------------------

from sys import exit
from math import sqrt

class TetTrace:
    "Data and methods related to tracing, interpolation, and point location on 3D
tet meshes"

    def __init__(self,meshdata):
        "Initializes the TetTrace class"

        self.meshdata = meshdata
        self.path = []
        self.end_vol_coord = []
        self.end_trace = 0
        self.start_coord = []
        self.end_coord = []
        self.omega = []
        self.total_path = 0.

    # -------------------------------------------------------------------

    def fieldVal(self, xyz_coord, field_data):
        "Finds the cell containing an arbitrary mesh point, and " + \
              "returns the LDFEM field value at that point"
        icell = 0
        end_coord = xyz_coord
        start_coord = self.centroid(icell)
        i = self.trackData(icell, start_coord, end_coord)
        return self.interp(field_data)

    # -------------------------------------------------------------------

    def interp(self, field_data):
        "Interpolates to find LDFEM field value inside a tet"
        icell = self.path[-1][0]
```

```
            x0 = field_data[icell][0]
            x1 = field_data[icell][1]
            x2 = field_data[icell][2]
            x3 = field_data[icell][3]
            return  ( x0*self.end_vol_coord[0] + x1*self.end_vol_coord[1] +
                      x2*self.end_vol_coord[2] + x3*self.end_vol_coord[3] )

        # ------------------------------------------------------------

        def cellNumb(self, xyz_coord):
            "Finds the cell containing an arbitrary mesh point"
            icell = 0
            end_coord = xyz_coord
            start_coord = self.centroid(icell)
            i = self.trackData(icell, start_coord, end_coord)
            return self.path[-1][0]

        # ------------------------------------------------------------

        def trackData(self, start_cell, start_coord, end_coord):
            "Calculates the cell numbers and path lengths in a ray"

            self.path = []
            self.end_cell = -999
            self.end_vol_coord = []
            self.end_trace = 0
            self.start_coord = start_coord[:]
            self.end_coord = end_coord[:]

            self.omega = [ end_coord[0]-start_coord[0],
                           end_coord[1]-start_coord[1],
                           end_coord[2]-start_coord[2] ]
            self.total_path = sqrt(self.omega[0]**2 + self.omega[1]**2 +
                                   self.omega[2]**2)
            if (self.total_path > 0.):
                xx = 1./self.total_path
                self.omega[0]=xx*self.omega[0]; self.omega[1]=xx*self.omega[1]
                self.omega[2]=xx*self.omega[2]

            icount = -1
            exit_coord = self.start_coord[:]
            next_cell = start_cell
            # walk through the ray one tet at a time.
            while (not self.end_trace):
                icount = icount+1
                if (icount >= self.meshdata.ncells-1):
                    print "Error: cells in track exceeds total cells in mesh"
                    exit()
                entry_coord=exit_coord[:]
                exit_coord = []
                current_cell = next_cell
                next_cell, exit_coord = self.cell_path(current_cell, entry_coord)
                # print self.path[-1] #list the cell numbers and path lengths for debug
            # Perform some error checking.
            x = 0.
            for i in self.path:
                if (i[0] < 0) or (i[0] >= self.meshdata.ncells):
                    print "Error: cell index out of range in trackData"
                    exit()
                x = x + i[1]
            if (abs(1. - x/self.total_path ) > 1.e-6 ):
                print "Error: total path length in error in trackData"
                exit()
            return len(self.path)

        # ------------------------------------------------------------
```

```
    def cell_path(self, current_cell, entry_coord):
        "Returns exiting coordinates and adjacent cell of a track through the
current_cell"

        # ** Note that omega must have unit magnitude. **

        # Check to see if the trace has exited the geometry
        if (current_cell == self.meshdata.bdry_flag[0]):
            self.end_trace = 1 #true
            return (-999, (0., 0., 0.) )

        # Check to see if the track terminates inside this cell.
        # Note that for computational efficiency if all the rays
        # terminate at one point, once the cell for that point is
        # found there is no need to perform repeated searches here.
        # A production algorithm would store this information
        # and avoid this test on every cell if the geometry permitted.
        vc = self.vol_coord(self.end_coord, current_cell)
        incell = 1 #true
        for i in vc:
            if (i < 0.) or (i > 1.):
                incell = 0 #false
                break
        if (incell):
            cell_path = sqrt( (entry_coord[0]-self.end_coord[0])**2 + \
                              (entry_coord[1]-self.end_coord[1])**2 + \
                              (entry_coord[2]-self.end_coord[2])**2 )
            self.path.append( (current_cell, cell_path) )
            self.end_vol_coord = vc
            self.end_trace = 1
            return (-999, (0., 0., 0.) )

        else:
            # Find the path length in this cell and the exiting face.
            cell_path = 1.e15
            outface = -999
            for i in range(self.meshdata.nfaces):
                xomp = []
                i0 = self.meshdata.face_node[i][0]
                n0 = self.meshdata.cell_node[current_cell][i0]
                x0 = self.meshdata.node_coord[n0][0]
                y0 = self.meshdata.node_coord[n0][1]
                z0 = self.meshdata.node_coord[n0][2]
                xomp.append(x0 - entry_coord[0])
                xomp.append(y0 - entry_coord[1])
                xomp.append(z0 - entry_coord[2])
                avect = self.meshdata.avect(current_cell,i)
                omega_dot_a = avect[0]*self.omega[0] + \
                              avect[1]*self.omega[1] + avect[2]*self.omega[2]
                if (omega_dot_a > -1.e-30):
                    omega_dot_a = max(omega_dot_a, 1.e-30)
                    xomp_dot_a = avect[0]*xomp[0] + \
                                 avect[1]*xomp[1] + avect[2]*xomp[2]
                    if (xomp_dot_a > -1.e-30):
                        xomp_dot_a = max(xomp_dot_a, 1.e-30)
                        xx = xomp_dot_a/omega_dot_a
                        if (xx < cell_path):
                            outface = i
                            cell_path = xx
            if (outface < 0):
                print "Error: unable to find an exiting face"
                exit()
            exit_coord = (entry_coord[0] + cell_path*self.omega[0],
                          entry_coord[1] + cell_path*self.omega[1],
                          entry_coord[2] + cell_path*self.omega[2])
            self.path.append( (current_cell, cell_path) )
            next_cell = self.meshdata.adjcell(current_cell,outface)[0]
```

```
        return (next_cell, exit_coord)

    # ------------------------------------------------------------------

    def centroid(self,icell):
        "Returns the centroid of cell icell"
        n0 = self.meshdata.cell_node[icell][0]
        n1 = self.meshdata.cell_node[icell][1]

        n2 = self.meshdata.cell_node[icell][2]
        n3 = self.meshdata.cell_node[icell][3]
        x0 = self.meshdata.node_coord[n0][0]
        y0 = self.meshdata.node_coord[n0][1]
        z0 = self.meshdata.node_coord[n0][2]
        x1 = self.meshdata.node_coord[n1][0]
        y1 = self.meshdata.node_coord[n1][1]
        z1 = self.meshdata.node_coord[n1][2]
        x2 = self.meshdata.node_coord[n2][0]
        y2 = self.meshdata.node_coord[n2][1]
        z2 = self.meshdata.node_coord[n2][2]
        x3 = self.meshdata.node_coord[n3][0]
        y3 = self.meshdata.node_coord[n3][1]
        z3 = self.meshdata.node_coord[n3][2]
        return ( 0.25*(x0+x1+x2+x3),
                 0.25*(y0+y1+y2+y3),
                 0.25*(z0+z1+z2+z3) )

    # ------------------------------------------------------------------

    def vol_coord(self, xyz_coord, icell):
        "Returns the icell based volume coordinates of a point"
        n0 = self.meshdata.cell_node[icell][0]
        n1 = self.meshdata.cell_node[icell][1]
        n2 = self.meshdata.cell_node[icell][2]
        n3 = self.meshdata.cell_node[icell][3]
        x0 = self.meshdata.node_coord[n0][0]
        y0 = self.meshdata.node_coord[n0][1]
        z0 = self.meshdata.node_coord[n0][2]
        x1 = self.meshdata.node_coord[n1][0]
        y1 = self.meshdata.node_coord[n1][1]
        z1 = self.meshdata.node_coord[n1][2]
        x2 = self.meshdata.node_coord[n2][0]
        y2 = self.meshdata.node_coord[n2][1]
        z2 = self.meshdata.node_coord[n2][2]
        x3 = self.meshdata.node_coord[n3][0]
        y3 = self.meshdata.node_coord[n3][1]
        z3 = self.meshdata.node_coord[n3][2]

        amat =  [ [ [], [], [] ],
                  [ [], [], [] ],
                  [ [], [], [] ] ]

        amat[0][0] = x1 - x0
        amat[0][1] = x2 - x0
        amat[0][2] = x3 - x0

        amat[1][0] = y1 - y0
        amat[1][1] = y2 - y0
        amat[1][2] = y3 - y0

        amat[2][0] = z1 - z0
        amat[2][1] = z2 - z0
        amat[2][2] = z3 - z0

        bvec = [ 0., 0., 0. ]

        bvec[0] = xyz_coord[0] - x0
```

```
        bvec[1] = xyz_coord[1] - y0
        bvec[2] = xyz_coord[2] - z0

        self.ge3(amat,bvec)

        xx = []
        xx.append(1. - bvec[0] - bvec[1] -bvec[2])
        xx.append(bvec[0])
        xx.append(bvec[1])
        xx.append(bvec[2])

        return xx

    # ------------------------------------------------------------------

    def ge3(self, amat, bvec):
        "Solves a 3x3 linear system using Crout's method with partial pivoting"

        # This is a general procedure that would likely
        # be obtained from a optimized linear algebra library in a
        # production code.

        # j = 0

        imax  = 0
        colmx = abs(amat[0][0])

        dum = abs(amat[1][0])
        if (dum >= colmx):
            imax = 1
            colmx = dum

        dum = abs(amat[2][0])
        if (dum >= colmx):
            imax = 2
            colmx = dum

        if (imax != 0):
            dum = amat[imax][0]
            amat[imax][0] = amat[0][0]
            amat[0][0] = dum
            dum = amat[imax][1]
            amat[imax][1] = amat[0][1]
            amat[0][1] = dum
            dum = amat[imax][2]
            amat[imax][2] = amat[0][2]
            amat[0][2] = dum
            dum = bvec[imax]
            bvec[imax] = bvec[0]
            bvec[0] = dum

        if (amat[0][0] == 0.):
            print "Error: zero pivot in ge3"
            exit()

        amat[0][0] = 1./amat[0][0]
        amat[1][0] = amat[1][0]*amat[0][0]
        amat[2][0] = amat[2][0]*amat[0][0]

        # j = 1

        amat[1][1] = amat[1][1] - amat[1][0]*amat[0][1]
        amat[2][1] = amat[2][1] - amat[2][0]*amat[0][1]

        imax = 1
        colmx = abs(amat[1][1])
```

```
        dum = abs(amat[2][1])
        if (dum >= colmx):
            imax = 2
            colmx = dum

        if (imax != 1):
            dum = amat[imax][0]
            amat[imax][0] = amat[1][0]
            amat[1][0] = dum
            dum = amat[imax][1]
            amat[imax][1] = amat[1][1]
            amat[1][1] = dum
            dum = amat[imax][2]
            amat[imax][2] = amat[1][2]
            amat[1][2] = dum
            dum = bvec[imax]
            bvec[imax] = bvec[1]
            bvec[1] = dum

        if (amat[1][1] == 0.):
            print "Error: zero pivot in ge3"
            exit()

        amat[1][1] = 1./amat[1][1]
        amat[2][1] = amat[2][1]*amat[1][1]

        # j = 2

        amat[1][2] = amat[1][2] - amat[1][0]*amat[0][2]
        amat[2][2] = amat[2][2] - amat[2][0]*amat[0][2] - amat[2][1]*amat[1][2]

        if (amat[2][2] == 0.):
            print "Error: zero pivot in ge3"
            exit()

        amat[2][2] = 1./amat[2][2]

        # forward and backward substitution

        bvec[1] =   bvec[1]-amat[1][0]*bvec[0]

        bvec[2] = ( bvec[2]-amat[2][0]*bvec[0]-amat[2][1]*bvec[1]
)*amat[2][2]
        bvec[1] = ( bvec[1]                    -amat[1][2]*bvec[2])*amat[1][1]
        bvec[0] = ( bvec[0]-amat[0][1]*bvec[1]-amat[0][2]*bvec[2])*amat[0][0]

        return 1

#-----------------------------------------------------------------------
#                       end of trace.py
#-----------------------------------------------------------------------

#  Sn Angular Quadrature Data ---
#  Set Name - Triangle Chebychev Lobatto, 18 angles

sph_degree
     1
end_sph_degree

#  ang#             mu             eta              zi              wgt
quadrature
      1 -1.0000000E+00  0.0000000E+00  0.0000000E+00  8.3333333E-02
      2 -4.4721360E-01 -8.2634298E-01 -3.4228247E-01  5.2083333E-02
      3 -4.4721360E-01 -3.4228247E-01 -8.2634298E-01  5.2083333E-02
      4  4.4721360E-01 -8.2634298E-01 -3.4228247E-01  5.2083333E-02
      5  4.4721360E-01 -3.4228247E-01 -8.2634298E-01  5.2083333E-02
      6 -4.4721360E-01  8.2634298E-01 -3.4228247E-01  5.2083333E-02
```

```
       7 -4.4721360E-01  3.4228247E-01 -8.2634298E-01  5.2083333E-02
       8 -4.4721360E-01 -8.2634298E-01  3.4228247E-01  5.2083333E-02
       9 -4.4721360E-01 -3.4228247E-01  8.2634298E-01  5.2083333E-02
      10 -4.4721360E-01  8.2634298E-01  3.4228247E-01  5.2083333E-02
      11 -4.4721360E-01  3.4228247E-01  8.2634298E-01  5.2083333E-02
      12  4.4721360E-01 -8.2634298E-01  3.4228247E-01  5.2083333E-02
      13  4.4721360E-01 -3.4228247E-01  8.2634298E-01  5.2083333E-02
      14  4.4721360E-01  8.2634298E-01 -3.4228247E-01  5.2083333E-02
      15  4.4721360E-01  3.4228247E-01 -8.2634298E-01  5.2083333E-02
      16  1.0000000E+00  0.0000000E+00  0.0000000E+00  8.3333333E-02
      17  4.4721360E-01  8.2634298E-01  3.4228247E-01  5.2083333E-02
      18  4.4721360E-01  3.4228247E-01  8.2634298E-01  5.2083333E-02
end_quadrature


#-----------------------------------------------------------------------
#
# File: aquad.py
#
#       Angular quadrature related methods and data. Any angular
#       quadrature can be used. These are the "ordinates" in the
#       discrete ordinates (Sn) method of angular discretization.
#
#
# Author: John McGhee, Radion Technologies
#
# Date  : 19 Feb 2004

#-----------------------------------------------------------------------
#
# $Id: aquad.py,v 1.7 2004/03/05 17:19:58 mcghee Exp $
#
#-----------------------------------------------------------------------

from sys import exit

class Quadrature:
    "Provides information on the angular quadrature."

    def __init__(self,quad_file):
          "Reads in the angular quadrature from disk."


          from string import split, atof, atoi, strip
          from sph import sphfun

          # Setup for read of geometry data from disk.
          self.name=quad_file
          f = open(quad_file)
          record = f.readline()

          # Read in the desired spherical harmonics degree:
          while (record and strip(record) != "sph_degree" ):
              record = f.readline()
          if (record == ""):
              print "Error: unable to find sph_degree keyword on quadrature file
%s\n" % quad_file
              exit()
          record = f.readline()
          self.sph_degree = atoi(record)
          if (self.sph_degree < 1):
              print "Error: spherical harmonics degree (lmax) must be >= 1"
              exit()

          # Read in the quadrature points from disk.
          while (record and strip(record) != "quadrature" ):
              record = f.readline()
```

```
            if (record == ""):
                print "Error: unable to find quadrature keyword on quadrature file
    %s\n" % quad_file
                exit()
            self.quad = []
            record = f.readline()
            while (record and strip(record) != "end_quadrature"):
                j = split(record)
                if (len(j) != 5):
                    print "Error: wrong number of entries on quadrature data line"
                    exit()
                self.quad.append(map(atof,j[1:5]))
                record = f.readline()
            if (record == ""):
                print "Error: unable to find end_quadrature keyword on quadrature
    file %s\n" % quad_file
                exit()
            self.nang = len(self.quad)

            # indexes between from the moment ordinal index and the (l,m) index
            self.nmom = (self.sph_degree+1)**2
            self.index_lm = []
            self.index_kk = {}
            k = -1
            for i in range(self.sph_degree+1):
                for j in range(-i,i+1):
                    k = k+1
                    x = (i,j)
                    self.index_lm.append(x)
                    self.index_kk[x] = k

            # This demo uses the "standard" method
            # of conversion from discrete to moment and visa versa.
            # A "Galerkin" or other more advanced mappping
            # method may be useful for certain problem types with highly
            # forward peaked scattering. These advanced methods
            # would be implemented here.

            # discrete to moment array.
            self.dis2mom_data = []
            for imom in range(self.nmom):
                x = []
                l = self.index_lm[imom][0]
                m = self.index_lm[imom][1]
                for kk in range(self.nang):
                    omega = self.qdpt(kk)
                    x.append( omega[3]*sphfun(l, m, omega[0], omega[1],
                                              omega[2]) )
                self.dis2mom_data.append(x)

            # moment to discrete array.
            self.mom2dis_data = []
            for kk in range(self.nang):
                omega = self.qdpt(kk)
                x = []
                for imom in range(self.nmom):
                    l = self.index_lm[imom][0]
                    m = self.index_lm[imom][1]
                    x.append( sphfun(l, m, omega[0], omega[1], omega[2]) )
                self.mom2dis_data.append(x)

        # ------------------------------------------------------------------

        def qdpt(self,kk):
            "Returns quadrature data for a single angle"
            if ( kk < 0 ) or ( kk >= self.nang ):
                print "Error: angle index out of range."
```

```
            exit()
        return self.quad[kk]

    # ------------------------------------------------------------------

    def dis2mom(self,kk):
        "Returns a column of the discrete to moment array"
        if (kk < 0) or (kk > self.nang):
            print "Error: kk index out of range in dis2mom"
            exit()
        x = []
        for imom in range(self.nmom):
            x.append(self.dis2mom_data[imom][kk])
        return x

    # ------------------------------------------------------------------

    def mom2dis(self,kk):
        "Returns a row of the moment to discrete array"
        if (kk < 0) or (kk > self.nang):
            print "Error: kk index out of range in mom2dis"
            exit()
        return self.mom2dis_data[kk]

#-----------------------------------------------------------------------
#                        end of aquad.py
#-----------------------------------------------------------------------
```

$Id: background_info.asc,v 1.4 2004/03/05 21:51:31 mcghee Exp $

1. Introduction

This is an overview of a demonstration code to be included with Radion Technologies utility patent for the application of deterministic transport methods to medical therapy and imaging problems,19 Mar 2004. The purpose of this code is to provide an example of the "preferred embodiment" of a 3D linearized Boltzmann transport (BTE) solver, and to demonstrate that the algorithm has been "reduced to practice". The demonstration code seeks to provide enough detail so that a worker "skilled in the art" could reproduce the significant aspects of the algorithm without difficulty. However, the demonstration code is not a complete production ready product, the scope is limited to demonstrating the key features involved.

2. Development of Mathematical Model

Particle transport is a physical process. This physical process can be mathematically modeled by a integro-partial differential equation (PDE).

In order to construct this model we make a series of more-or-less standard assumptions regarding the physical process of particle

transport. These assumptions are as follows:

-- particles are treated as points, no quantum mechanical wave effects are included. No wave effects such as polarization, refraction, or interference are included.

-- particles travel in straight lines between point collisions, particles are unaffected by external forces such as gravity or electrical and magnetic fields.

-- particle-particle interactions are neglected. The particle density is assumed to be low compared with the density of the surrounding medium. -- collisions are assumed to be instantaneous, scattering events are not delayed.

-- material properties are isotropic, there is no preferred direction of particle travel within a single material.

-- material properties are known a priori and are not a function of the particle distribution.

-- it is assumed that the particle density is sufficiently high that a mean value of the distribution adequately describes the solution.

Under these assumptions we construct what we refer to as the linearized Boltzmann Transport Equation (BTE). This equation is a hyperbolic

integro partial differential equation. The solution to this equation provides a distribution of particles in space, angle, and energy which
is an (hopefully accurate) approximation to the actual particle distribution associated with the physical transport process we are trying to model.
For charged particle problems a continuous-slowing-down (CSD) term in energy from the Boltzmann-Fokker-Planck (BFP) equation is included in the BTE.

We assume that only the steady state solution is of interest. The
BTE has a term included for time-dependent problems, but we elect not
to include this term in our formulation as the radiation equilibration times are very short compared to the time scales of interest for the present application. We have included this term on other projects without difficulty.

3. Development of Numerical Model

Given the mathematical model described above, a discrete approximation to that model is created which is amenable to numerical solution.
The algorithm we employ consists of a set of individual techniques extant in the transport literature for some time. However, these techniques are assembled into a novel and particularly effective combination for our application. The techniques used are:
-- multi-group discretization in energy, with a discontinuous differencing in energy of the BFP CSD term.
-- discontinuous finite element (DFEM) spatial differencing on an unstructured finite element mesh, and,
-- discrete ordinates (Sn) differencing in angle.

4. Defining (critical) features of demonstration code.

Equation solved:
-- a discretized approximation to the first order form of the
linearized BTE with an additional BFP CSD term for charged particles, a first order hyperbolic integro partial differential equation

Discretization:
-- multi-group energy differencing
-- unstructured finite element mesh

-- discontinuous finite element spatial differencing (DFEM)
-- discrete ordinates angular differencing (Sn)
-- spherical harmonics expansion of scattering source
-- produces a large, sparse, asymmetric linear system of equations
in a six dimensional phase space.
Solution technique:
-- Source iteration (Richardson iteration)
-- Sweep ordering and cycle breaking to produce lower triangular system --
Diffusion Synthetic Acceleration (DSA) of the source iteration
process.

Important features to be demonstrated:
-- multiple particle problems are supported
-- interpolation of solution on FE mesh.
-- ability to restart from previous solution
-- ability to control refinement in space, angle, and energy

Other significant novel aspects:
-- 3D lobatto-chebychev angular quadratures
-- ray tracing option to minimize ray effects for point sources, including
first, second, and subsequent scattered sources.
-- memory management techniques
-- ability to provide adjoint solutions
-- ability to accommodate a variety of material properties databases in a transparent and
modular fashion.
-- alternative solution methodologies - Krylov solvers.
-- parallelization: angular domain decomposition, sweep scheduling,
Krylov solvers

5. Key Advantages of the Method Described

Many techniques for discretizing the BTE have been developed over the years. The method described herein is an attractive solution algorithm because it provides a capability to capture the true transport solution everywhere in a problem with a minimal amount of computational effort. This method provides "the accuracy heretofore attributed exclusively to Monte Carlo, without the prohibitive computational cost." The major features which provide this capability are as

follows:

-- the Sn, multi-group method with arbitrary order anisotropic scatter is a convergent method. That is to say that it captures all the physics in the BTE and converges to the true transport solution as the computational mesh is refined in angle, space, and energy.

-- arbitrary finite element mesh accurately captures the true problem geometry with a minimum number of spatial elements

-- discontinuous finite element differencing captures discontinuities in solution and material properties and is 3rd order accurate, acceleratable, damped, and has the diffusion limit.

-- the method provides a complete solution everywhere in the problem, free from statistical noise. There is no need to guess areas of interest beforehand as with Monte Carlo techniques.

-- rapid solutions are possible, arbitrary mesh provides capability to add resolution where needed and yet remain coarse elsewhere. DFEM spatial differencing remains accurate even on coarse meshes.

-- iterative solution technique provides a means to utilize information from previously solved problems to minimize cost of subsequent perturbation analysis.

-- ability to control resolution in space, angle, and energy on a

problem by problem basis saves computational time by avoiding excess resolution in problems or problem regions which do not require it.

6. References

1) Wareing, T.A., J.M. McGhee, J.E. Morel, S.D. Pautz, "Discontinuous Finite Element Sn Methods on Three Dimensional Unstructured Grids", Nuclear Science and Engineering, Vol 138, 256-268 (2001).

2) Pautz, S.D., "An Algorithm for Parallel Sn Sweeps on Unstructured Meshes", Nuclear Science and Engineering, Vol 140, 111-136, (2002).

3) Lewis, E.E. and W.F. Miller, Jr. "Computational Methods of Neutron Transport", John Wiley and Sons, New York, 1984.

4) Zienkiewicz, O.C., and R. L. Taylor", "The Finite Element Method", Fourth Edition, McGraw-Hill Book Company, London, 1994

```
#-----------------------------------------------------------------------
#
# File: bte.py
#
#       Sets up and solves the Boltzmann transport equation for
#       a specific group, angle, and cell. This class is specific
#       to tets for simplicity of exposition. In a general purpose code
#       this class would likely be a virtual base class, with derived
#       classes for several different element types.
#
#
# Author: John McGhee, Radion Technologies
#
# Date  : 19 Feb 2004

#-----------------------------------------------------------------------
#
# $Id: bte.py,v 1.17 2004/03/05 17:19:58 mcghee Exp $
#
#-----------------------------------------------------------------------

from sys import exit

class BteEquation:
    "Forms the LD linear tet Boltzmann Transport Equation"
    cell_type = "tetrahedra"

    def __init__(self, geom, icell, icp, omega, sigt, beta, kk):
        "Constructs the left hand side of the LD linear tet Boltzmann Equation"

        self.ib=[]
        self.bvec = []
        self.icell = icell
        self.icp = icp
        self.n = 4
        self.qcsd = []
        self.sum_qcsd = 0.
```

```
self.gamma = 0.
self.psiE = 0.
self.kk = kk

# Form omega dot A for each face in the cell
x = []
for iface in range(geom.nfaces):
    xx = 0.
    yy = geom.avect(icell,iface)
    for j in range(geom.ndim):
        xx = xx + omega[j]*yy[j]
    xx = xx/12.
    x.append(xx)

a = []
b = []
c = []
for i in x:
    if ( i < -geom.dplimit):
        a.append(0.)
        b.append(abs(i))
        c.append(1)
    elif ( i > geom.dplimit):
        a.append(i)
        b.append(0.)
        c.append(0)
    else:
        a.append(0.)
        b.append(0.)
        c.append(0)

# Construct the removal term
r = sigt*geom.vol(icell)/20.

# Construct the CSD terms for charged particle problems.
# Charged particle problems introduce an additional unknown
# into the solution trial space, which introduces an
# additional column and row into the transport matrix. We
# elect eliminate the extra unknown from the matrix by cyclic
# reduction, thereby keeping the number of rows and columns in
# the transport matrix the same for both charged and neutral
# particle problem types.
if (icp):
    self.k1 = beta*geom.vol(icell)/4.
    self.k2 = 1./( 4.*(a[0]+a[1]+a[2]+a[3]) + \
                   sigt*geom.vol(icell)/3. + 4.*self.k1)
    xx = self.k1*self.k1*self.k2
    cp0 = xx + 0.2*self.k1
    cp1 = xx + 0.4*self.k1
else:
    cp0 = 0.
    cp1 = 0.

# Form the transport matrix for this cell
self.amat=[ [],[],[],[] ]
self.amat[0].append( x[0] + 2.*(a[1]+a[2]+a[3]) + 2.*r + cp1)
self.amat[0].append( x[0] + a[2]+a[3] + r + cp0 )
self.amat[0].append( x[0] + a[1]+a[3] + r + cp0 )
self.amat[0].append( x[0] + a[1]+a[2] + r + cp0 )

self.amat[1].append( x[1] + a[2]+a[3] + r + cp0 )
self.amat[1].append( x[1] + 2.*(a[0]+a[2]+a[3]) + 2.*r + cp1)
self.amat[1].append( x[1] + a[0]+a[3] + r + cp0 )
self.amat[1].append( x[1] + a[0]+a[2] + r + cp0 )

self.amat[2].append( x[2] + a[1]+a[3] + r + cp0 )
self.amat[2].append( x[2] + a[0]+a[3] + r + cp0 )
self.amat[2].append( x[2] + 2.*(a[0]+a[1]+a[3]) + 2.*r + cp1)
```

```
        self.amat[2].append( x[2] + a[0]+a[1] + r + cp0 )

        self.amat[3].append( x[3] + a[1]+a[2] + r + cp0 )
        self.amat[3].append( x[3] + a[0]+a[2] + r + cp0 )
        self.amat[3].append( x[3] + a[0]+a[1] + r + cp0 )
        self.amat[3].append( x[3] + 2.*(a[0]+a[1]+a[2]) + 2.*r  + cp1)

        # Construct the coefficients used to form the in-flow terms.
        self.yy=[]
        self.yy.append(
            ( 2.*b[1], 2.*b[2], 2.*b[3], 0., b[2], b[3],
              0., b[1], b[3], 0., b[1], b[2] ) )
        self.yy.append(
            ( 0., b[2], b[3], 2.*b[0], 2.*b[2], 2.*b[3],
              b[0], 0., b[3], b[0], 0., b[2] ) )
        self.yy.append(
            ( b[1], 0., b[3], b[0], 0., b[3], 2.*b[0],
              2.*b[1], 2.*b[3], b[0], b[1], 0.) )
        self.yy.append(
            ( b[1], b[2], 0., b[0], b[2], 0., b[0], b[1],
              0., 2.*b[0], 2.*b[1], 2.*b[2]) )

        self.incoming = ( c[1], c[2], c[3], c[0], c[2], c[3],
                          c[0], c[1], c[3], c[0], c[1], c[2] )

        # Construct the coefficients used to form the charged particle
        # in-flow terms.
        self.zz = []
        self.zz = [ 4.*b[0], 4.*b[1], 4.*b[2], 4.*b[3] ]

    # ------------------------------------------------------------

    def rhs(self, geom, qvec, soldata, solved, qcsddata, psiE):
        "Forms the right hand side of the LD linear tet Boltzmann Equation"

        # Construct the in-flow terms from the adjacent cells.
        xx = []
        j = -1
        for ivrtx in range(geom.nvrtx):
            for iface in range(geom.nfaces):
                if ( ivrtx == iface ):
                    continue
                else:
                    j = j+1
                    i = geom.adjvrtx(self.icell,ivrtx,iface)
                    if ( i == geom.bdry_flag):
                        # This demo assumes vacuum boundary conditions.
                        # Other boundary conditions such as reflective, periodic,
                        # albedo, or source conditions could be
                        # implemented here by setting the value of x
                        # appropriately.
                        x = 0.
                    else:
                        # Make sure that no un-set up-wind information
                        # is  being requested.
                        if ( ( not solved[ i[0] ] ) and ( self.incoming[j] ) ):
                            print "cell    = ", self.icell
                            print "adj cell= ", i[0]
                            print "Error: attempt to use data from unsolved cell"
                            exit()
                        x = soldata[i[0]][i[1]]
                    xx.append(x)
        self.bevc = []
        for i in range(geom.nvrtx):
            x = 0.
            for j in range(12):
                x = x + self.yy[i][j]*xx[j]
            self.bvec.append(x)
```

```
        # If this is a charged particle problem, create the terms
        # arising from the cyclic reduction of the CSD operator.
        if (self.icp):
            qcsd = qcsddata[self.kk][self.icell]
            self.gamma = 0.
            for iface in range(geom.nfaces):
                i = geom.adjcell(self.icell,iface)[0]
                if ( i == geom.bdry_flag[0]):
                    continue
                else:
                    self.gamma = self.gamma + psiE[i]*self.zz[iface]
            self.sum_qcsd = qcsd[0]+qcsd[1]+qcsd[2]+qcsd[3]
            xx = self.k1*self.k2*(self.sum_qcsd + self.gamma)
            cp0 =  xx + 0.2*(2.*qcsd[0]+qcsd[1]+qcsd[2]+qcsd[3])
            cp1 =  xx + 0.2*(2.*qcsd[1]+qcsd[0]+qcsd[2]+qcsd[3])
            cp2 =  xx + 0.2*(2.*qcsd[2]+qcsd[1]+qcsd[0]+qcsd[3])
            cp3 =  xx + 0.2*(2.*qcsd[3]+qcsd[1]+qcsd[2]+qcsd[0])
        else:
            cp0=0.; cp1=0.; cp2 = 0.; cp3 = 0.

        # Add in the fixed source, scattering source, CSD terms, et al.
        r = geom.vol(self.icell)/20.
        self.bvec[0] = self.bvec[0] + r*(2.*qvec[0] + qvec[1] +
                                         qvec[2] + qvec[3]) + cp0
        self.bvec[1] = self.bvec[1] + r*(2.*qvec[1] + qvec[0] +
                                         qvec[2] + qvec[3]) + cp1
        self.bvec[2] = self.bvec[2] + r*(2.*qvec[2] + qvec[1] +
                                         qvec[0] + qvec[3]) + cp2
        self.bvec[3] = self.bvec[3] + r*(2.*qvec[3] + qvec[1] +
                                         qvec[2] + qvec[0]) + cp3

# ----------------------------------------------------------------

def ludecomp(self):
    "Performs lu decomposition in place using Crout's Method"

    # This method would likely be replaced in a production code by
    # a call to a highly optimized linear algebra library
    # function.

    itmp = []
    self.ib = []
    for i in range(self.n):
        itmp.append(i)
        self.ib.append(i)

    # Loop over columns
    for j in range(self.n-1):

        # Find pivot
        imax = j
        colmx = abs( self.amat[j][j] )
        for i in range(j+1,self.n):
            dum = abs(self.amat[i][j])
            if (dum > colmx):
                imax = i
                colmx = dum

        # Pivot this row, if required
        if (imax != j):
            for jj in range(self.n):
                dum = self.amat[imax][jj]
                self.amat[imax][jj] = self.amat[j][jj]
                self.amat[j][jj] = dum
            dum = itmp[imax]
            itmp[imax] = itmp[j]
            itmp[j] = dum
```

```
            # Perform Crout's method algebra
            if ( self.amat[j][j] == 0. ):
                print "Error: zero pivot encountered in ludecomp"
                exit()
            self.amat[j][j] = 1./self.amat[j][j]
            for i in range(j+1,self.n):
                self.amat[i][j] = self.amat[i][j]*self.amat[j][j]
            for jj in range(j+1):
                for ii in range(jj+1,self.n):
                    self.amat[ii][j+1] = self.amat[ii][j+1] - \
                                         self.amat[ii][jj]*self.amat[jj][j+1]

        # Invert the last diagonal element
        ii = self.n-1
        if (self.amat[ii][ii] == 0.):
            print "Error: zero pivot encountered in ludecomp"
            exit()
        self.amat[ii][ii] = 1./self.amat[ii][ii]

        # Store the permutation vector
        for i in range(self.n):
            self.ib[itmp[i]] = i

    # ----------------------------------------------------------------

    def fbsub(self):
        "Given a lu decomp and a source, solves the system via fwd and back
substitution"

        # This method would likely be replaced in a production code by
        # a call to a highly optimized linear algebra library
        # function.

        # permute the source
        x = self.bvec[:]
        for i in range(self.n):
            self.bvec[ self.ib[i] ] = x[i]
        # Forward substitution
        for j in range(self.n-1):
            for i in range(j+1,self.n):
                self.bvec[i] = self.bvec[i] - self.amat[i][j]*self.bvec[j]
        # Back substitution
        for j in range(self.n-1,-1,-1):
            self.bvec[j] = self.bvec[j]*self.amat[j][j]
            for i in range(j-1,-1,-1):
                self.bvec[i] = self.bvec[i] - \
                self.amat[i][j]*self.bvec[j]

        # If this is a charged particle problem, then compute
        # the energy slope term in the solution and create the
        # CSD source term for the next group.
        if (self.icp):
            self.psiE = self.k2*(self.sum_qcsd + self.gamma - self.k1*(
                self.bvec[0]+ self.bvec[1] + self.bvec[2] + self.bvec[3]))
            self.qcsd = [ self.k1*(self.bvec[0] - self.psiE),
                          self.k1*(self.bvec[1] - self.psiE),
                          self.k1*(self.bvec[2] - self.psiE),
                          self.k1*(self.bvec[3] - self.psiE) ]

#-----------------------------------------------------------------------
#                       end of bte.py
#-----------------------------------------------------------------------
#-----------------------------------------------------------------------
#
# File: edit.py
#
#       Defines procedures for producing edits.
```

```
#
#       Once a solution is calculated, links to any number of various
#       post-processing utilities can be generated easily. A sample
#       visualization link file generator is included here for
#       debug and demonstration purposes.
#
# Author: John McGhee, Radion Technologies
#
# Date  : 19 Feb 2004

#-----------------------------------------------------------------------
#
# $Id: edit.py,v 1.6 2004/03/05 21:51:31 mcghee Exp $
#
#-----------------------------------------------------------------------

def gmvlink(geom,  momdata, outfile):
    "Writes GMV format visualization link file"

    # GMV is a publicly available visualization utility
    # that can be obtained from Los Alamos National Laboratory,
    # Los Alamos, NM. http://www-xdiv.lanl.gov/XCM/gmv/

    from string import join
    from time import ctime, time
    from math import sqrt

    # Write the GMV visualization file to disk.
    print "\nWriting GMV format visualization link file to disk....."

    # Open the output file.
    f = open(outfile, "w")

    # Write magic cookie.
    f.write("gmvinput ascii   \n")

    # Write the nodes block
    data_per_line = 10
    f.write("\nnodes   " + str(geom.nnodes) + "\n")
    for j in range(geom.ndim):
        ic = 0
        for i in geom.node_coord:
            ic = ic+1
            f.write("%e " % i[j])
            if (ic % data_per_line == 0):
                f.write("\n")
        if (ic % data_per_line != 0):
            f.write("\n")

    # Write the cells block
    f.write("\ncells   " + str(geom.ncells) + "\n" )
    for i in geom.cell_node:
        record = ""
        f.write("tet      4\n")
        j =[]
        for k in i:
            j.append(k+1)
        record = join(map(str,j)) + "\n"
        f.write(record)

    # Write the material block
    f.write("\nmaterial" + " " + str(geom.nregions) + "  0\n")
    tmp=[]
    for j in range(geom.nregions):
        tmp.append("mat#-" + str(j))
    f.write(join(tmp) + "\n")
    ic = 0
    for j in geom.cell_region:
```

```
            ic = ic+1
            k = j+1
            f.write("%i " %  k)
            if (ic % data_per_line == 0):
                f.write("\n")
        if (ic % data_per_line != 0):
            f.write("\n")

        # Write the variable block. Variables printed here are fluxes and
        # currents for all energy groups. Other variables of specific
        # interest to a particular problem or user can easily be included if
        # required.
        moms = (0, 2, 3, 1) # reorder currents to x,y,z
        f.write("\nvariable\n")
        nmom = min(4,len(momdata))
        ngroups = len(momdata[0])
        for ig in range(ngroups):
            if (ngroups > 1):
                chdum="-g" + str(ig)
            else:
                chdum=""
            for ii in range(nmom):
                imom = moms[ii]
                if (imom == 0):
                    xnorm = 1.
                    f.write("phi0%s 0 \n" % chdum)
                elif (imom == 1):
                    xnorm = 1./sqrt(3.)
                    f.write("Jz%s 0 \n" % chdum)
                elif (imom == 2):
                    xnorm = 1./sqrt(3.)
                    f.write("Jx%s 0 \n" % chdum)
                else:
                    xnorm = 1./sqrt(3.)
                    f.write("Jy%s 0 \n" % chdum)
                ic = 0
                for i in range(geom.ncells):
                    ic = ic+1
                    xx = 0.
                    j = momdata[imom][ig][i]
                    kk = 0
                    for k in j:
                        kk = kk+1
                        xx = xx + k
                    xx = xnorm*xx/float(kk)
                    f.write("%e " %  xx)
                    if (ic % data_per_line == 0):
                        f.write("\n")
                if (ic % data_per_line != 0):
                    f.write("\n")
        f.write("\nendvars\n")

        # Write the end of file keyword
        f.write("\nendgmv  \n\n")

        # Close the files.
        f.close()


        # Echo success.
        print "GMV visualization link written to file \"%s\"" % outfile
        t = ctime(time())
        print t, "\n"

#--------------------------------------------------------------------
#                        end of edit.py
#--------------------------------------------------------------------
```

```
Sample of standard out from the Frost transport solver.

$Id: frost.out_std,v 1.4 2004/03/10 04:24:22 mcghee Exp $

------------------------------------------------------------

FROST - FiRst Order Sn Transport
        Radion Technologies

Mesh data successfully read from file mesh.inp
  ncells      =  862
  nnodes      =  207
  nbdry_faces=  200
  Calculation of mesh geometry data complete, checking results....
  Mesh data checks passed.

Problem Description
  ngroups=  2
  nang    =  18

  nmat    =  2
  nscxs   =  2
  nmom    =  4
  Neutral Particle Problem

Calculating solution in group  0
  0 delta=  420.854728454
  1 delta=  3.38884624607
  2 delta=  0.027504091683
  3 delta=  0.000224887198983
  4 delta=  1.85051804069e-06
  5 delta=  1.53046175519e-08
  6 delta=  1.27064696158e-10
  7 delta=  1.05796122693e-12
  8 delta=  8.82732294317e-15
Group  0 converged.

Calculating solution in group  1
  0 delta=  0.40013959077
  1 delta=  0.00127508909827
  2 delta=  4.07191272391e-06
  3 delta=  1.30328388021e-08
  4 delta=  4.18163292129e-11
  5 delta=  1.3448187674e-13
Group  1 converged.


Writing GMV format visualization link file to disk.....
GMV visualization link written to file "gmv.out"
Fri Mar  5 14:39:04 2004

Dose for entire mesh:  0.00540526704499

Results in group 0, moment 0, (x=0.5, y=0.6, z=0.7)
  phi0=  0.49361140853 , in cell number:  646

Balance Table--
  leakage        =  [0.71825356732193335, 0.018107227200817064]
  other removal  =  [0.2817464257358761, 0.019458961361966235]
  total source   =  [0.99999999999999756, 0.037566190098116937]
  balance        =  [6.9421881576658961e-09, 4.0870091755351723e-08]

Transport Solution Complete.

------------------------------------------------------------
```

```
#!/usr/bin/python
# ----------------------------------------------------------------------
#
# File: frost.py
#
#       FROST - FiRst Order Sn Transport.
#
#       A simple example of a 3D numerical solver for the first order
#       form of the Boltzmann Transport
#       Equation (BTE) using a linear discontinuous finite element
#       spatial discretization, a discrete ordinates angular
#       discretization, and a multi-group energy discretization.
#       Written in Python ( http://http://www.python.org/ ).
#       Anisotropic scattering, multiple-material, multiple-particle,
#       and charged-particle problem types are supported. Also
#       provides an option for a first scattered distributed source
#       solution demo mode. Any type of neutral or charged particle may be
#       transported including neutrons, photons, protons, and electrons.
#
#       This implementation is not optimized for computational
#       efficiency but rather seeks to provide a relatively simple and
#       clear example of the principles composing the algorithm. This
#       demo is based on a tetrahedral finite element mesh for
#       purposes of simplification, but the
#       techniques used herein can and have been applied on other mesh
#       types as well. Like-wise the solution is assumed to be
#       linear within a tet, but higher order solution trial spaces
#       can be applied as well in a standard finite element manner.
#
#       Method of Solution:
#       The method of solution demonstrated herein consists of
#       ordering the cell equations into
#       a block lower triangular form, sweeping the mesh for each
#       discrete ordinate, and then computing the scattering source via
#       source iteration. Other solution methods are possible. Of note
#       is recent work at Los Alamos on Krylov based solvers rather
#       than source iteration. Various parallelization opportunities
#       are associated with these solvers that may make them an
#       attractive alternative in the future.
#
#       Source code size and performance:
#       This demo code is approximately 2,400 lines. It runs the
#       included demo problem in approximately 160 seconds on a 2.5
#       Mhz desktop PC. A Fortran code in production use which
#       provides more efficiency, flexibility, error checks, a user
#       friendly interface, etc. is approximately
#       50,000 lines and performs the same calculation in a fraction of
#       second.
#
#       Parallelization:
#       This is a serial code. Various methods for parallelization are known
#       and have been implemented in production versions of this
#       algorithm. In general parallelization does not substantially
#       alter the structure of the algorithm and we elect not to
#       include parallelization herein in the interest of clarity.
#       However, comments are inserted where appropriate to indicate
#       where opportunities for parallelization exist.
#
#       Disclaimer:
#       The purpose of this code is to demonstrate the essential
#       characteristics of the solution algorithm. It is not a
#       production ready solver. As such, the code has been subjected
#       to a minimal set of testing and debug procedures. The results
#       compare favorably with other more well tested solvers. It produces
#       qualitatively reasonable results and passes fundamental tests
#       for accuracy such as particle balance. However, it has not
```

```
#       been tested extensively or thoroughly debugged on a wide
#       variety of problems. Caveat Emptor.
#
#       A note on Python syntax:
#       Indentation is significant! Indentation is used to denote
#       different blocks of code such as bodies of functions,
#       conditionals, loops, and classes. Care should be exercised
#       when copying or reformatting Python source code to avoid
#       unintentionally altering the functionality of
#       the code.
#
# Usage: ./frost.py [fsds]
#
#        The optional argument "fsds" triggers a first-scattered
#        distributed source solution mode option.
#
#        Input is read in from three files:
#          -- aquad.inp, angular quadrature data.
#          -- mesh.inp, problem geometry and computational mesh.
#          -- matprop.inp, material properties
#
#        Fixed sources are defined in a the file "fsrc.py". Two
#        sources are provided, both isotropic in group 0 with
#        strength of 1. One source is uniformly distributed, and
#        the other is a point source located at (0,0,0).
#
# Author: John McGhee, Radion Technologies
# Date  : 19 Feb 2004
#
# ------------------------------------------------------------------
#
# $Id: frost.py,v 1.41 2004/03/10 17:17:26 mcghee Exp $
#
# ------------------------------------------------------------------

# Import methods and classes used to solve the transport equation.
from sys   import exit, argv
from geom  import Geometry
from bte   import BteEquation
from aquad import Quadrature
from fsrc  import FixedSource
from matp  import MaterialProps
from sord  import SolutionOrder
from edit  import gmvlink
from scat  import self_scatter, in_scatter
from fsds  import Fsds
from trace import TetTrace

# Write a header.
print "\nFROST - FiRst Order Sn Transport"
print "        Radion Technologies\n"

# Get the command line parameters
i = len(argv)
if (i == 1):
    fsds_prob = 0
elif ( i == 2 ):
    if (argv[1] == "fsds"):
        fsds_prob = 1
    else:
        print "Error: unrecognized command line parameter \"",argv[1],"\""
        print "Valid options are: no-argument | fsds"
        print
        exit()

# Set convergence criteria and max allowed iterations for the source
# iteration process. Ordinarily this would be set by the user on a by
# problem basis.
```

```
eps      = 1.e-12
iter_max = 20

# Read mesh geometry. Tets are assumed here, but
# other geometries are possible. Higher order solution trial
# spaces (i.e. sub-parametric, or p-refinement) can also
# be used in the standard finite element manner.
meshdata = Geometry("mesh.inp")

# Read angular quadrature data. Any angular quadrature
# can be used depending on the requirements of the problem at hand.
# For example, Radion has developed special 3D lobatto-chebychev sets
# for use with problems which contain plane wave sources.
qdata =Quadrature("aquad.inp")

# Read multi-group material properties. Multiple material problems
# with anisotropic up, down, and within-group scatter are supported.
# Scattering cross sections are provided in terms of an expansion in
# Legendre coefficients. Multiple particle type problems can be handled
# transparently by the multi-group method by simply assigning
# different groups to different particle types. For this demo code
# there is assumed to be a one-to-one
# correspondence between the materials and the mesh regions, a
# production version would provide more flexibility in material
# assignment. Also of note, adjoint solutions to the transport

# equation are easily provided for by performing simple modifications
# to the material properties and their indexes. There are normally
# only a few tens of materials in a transport problem, and many
# thousands of computational cells. Considerable memory savings can
# be achieved by storing the material properties by material
# and assigning material properties to cells only on an as-needed
# basis.
matprop = MaterialProps("matprop.inp")

# Setup fixed volume source. Full anisotropic, spatially varying, multi-group
# sources are supported. For this demo code the fixed source values
# are provided by a pre-defined function. In a production code,
# provisions would be made to allow the user to specify the fixed
# source characteristics in detail as part of the problem setup.
fsdata = FixedSource(qdata, meshdata, matprop)

# Setup fixed boundary source.
# This demo assumes volume sources only, but boundary sources
# can be handled in a similar manner to volume sources. Boundary
# sources are entered into the solution algorithm in the rhs method of
# the BteEquation class.

# Perform analytic un-collided flux calculation.
# If desired an un-collided solution component can be calculated using
# analytic, high Sn order, or other techniques. This un-collided
# component can then be used to compute a first collided source to
# initiate a follow on calculation to complete the solution. Among
# other things this is often useful as a ray-effect mitigation
# technique. Second and/or subsequent collided solution components and
# their associated scattering sources can also be
# calculated and combined in a similar manner if desired.
if (fsds_prob):
    if (matprop.icp):
        print "Error: fsds option is not supported for charged particle problems"
        exit()
    fsdsdata = Fsds(matprop, meshdata, qdata, fsdata)

# Create an array for storing the angular source.
qvec = [ 0., 0., 0., 0. ]

# Create angular solution storage array.
soldata = []
```

```
for i in xrange(meshdata.ncells):
    soldata.append([0., 0., 0., 0.])

# Create the moment solution storage array. For a reduction in memory
# at the expense of runtime, this array (and others) could be stored
# on disk and read into memory group-by-group as needed. This array
# can be initialized from a previous solution stored on disk, which
# can save substantial computational work in the iterative solution
# process by providing a good starting guess. This is especially
# advantageous in problems such as perturbation analysis, where the
# solution is changing minimally from one analysis to the next.
momdata = []
for k in range(qdata.nmom):
    momdata.append( [] )
    for j in range(matprop.ngroups):
        x= []
        for i in xrange(meshdata.ncells):
            x.append([0., 0., 0., 0.])
        momdata[k].append(x)

# Create a tmp storage array for source iteration
oldmom = []
for k in range(qdata.nmom):
    oldmom.append( [] )
    for j in range(matprop.ngroups):
        x= []
        for i in xrange(meshdata.ncells):
            x.append([0., 0., 0., 0.])
        oldmom[k].append(x)

# If this is a charged particle problem, then the transport equation
# is altered to include the continuous-slowing-down (CSD) term from
# the Boltzmann-Fokker-Planck operator. We difference this term using
# a linear discontinuous treatment in energy which introduces an
# additional unknown into the solution trial space. Additional storage
# is required for the treatment of this unknown, which we set up below.
psiE  = []
qcsd  = [ [], [] ]
csdptr_g   = 1
csdptr_gm1 = 0
if (matprop.icp):
    for k in range(2):
        for j in range(qdata.nang):
            x= []
            for i in xrange(meshdata.ncells):
                x.append( [0., 0., 0., 0.] )
            qcsd[k].append(x)
    for i in xrange(meshdata.ncells):
        psiE.append(0.)

# Create storage arrays for a balance table.
abs = []
leakage = []
source = []
balance = []
csd_src = []
csd_rem = []
for j in range(matprop.ngroups):
    abs.append( 0. )
    leakage.append( 0.)
    source.append( 0.)
    balance.append( 0.)
    csd_src.append(0.)
    csd_rem.append(0.)

# Determine sweep ordering. For a reduction in memory at a cost of
# increased runtime, this could be computed on the fly for each angle
# inside of the source iteration loop. Also, a fine grained
```

```
# parallelization can be accomplished at this step by assignment of
# cells to a multiprocessor solution schedule based on a careful analysis
# of the graph associated with the problem mesh. This technique has been
# implemented in research and development prototypes.
swpdata = SolutionOrder(meshdata, qdata)

# Echo a few details to std out to identify the problem parameters.
print "Problem Description"
print "  ngroups= ", matprop.ngroups
print "  nang   = ", qdata.nang
print "  nmat   = ", matprop.nmat
print "  nscxs  = ", matprop.nscxs
print "  nmom   = ", qdata.nmom
if (matprop.icp):
    print "  Charged Particle Problem"
else:
    print "  Neutral Particle Problem"
if (fsds_prob):
    print "  First Scattered Distributed Source Problem"
    print "    nrays = ", fsdsdata.nrays
    print "    average cells per ray = %5.1f" % fsdsdata.ncells_per_ray
print

# If upscatter is present, then an additional "outer" iteration would
# be wrapped around the energy group loop at this point. This demo
# assumes downscatter only, so the outer iteration loop is not
# present. Also, for some classes of multi-particle problems it may be
# advantageous to solve the block of groups associated with each
# particle as a separate calculation in order to optimize the
# computation for each particle type.

# Loop over energy groups from high energy to low. We use the nuclear
# engineering convention for group numbers (ie. group 0 is the highest
# energy group, and group "ngroups" is the lowest energy group.
for ig in range(matprop.ngroups):
    print "Calculating solution in group ", ig
    delta  =  1.# a measure of the change in the solution from one
               # iteration to the next
    icount = -1 # the iteration counter

    # For charged particle problems there is a source from the CSD
    # term from the group above in energy. To save memory, we only
    # store the source from the preceding group and the current
    # group. Here we swap the pointers for the current and
    # preceding groups.
    x          = csdptr_gm1
    csdptr_gm1 = csdptr_g
    csdptr_g   = x

    # loop over the within-group scattering (self-scatter) source
    # until converged for this group.
    while (delta > eps):
        # Clear the balance table
        csd_src[ig] = 0.
        csd_rem[ig] = 0.
        leakage[ig] = 0.
        abs[ig]     = 0.
        source[ig]  = 0.
        # increment the iteration counter
        icount = icount + 1
        # trap a likely error condition to prevent infinite iteration.
        if (icount > iter_max):
            print "Error: maximum allowed number of inner iterations exceeded."
            print "Iterations      : ", icount
            print "Current delta   : ", delta
            print "Convergence crit: ", eps
            break
```

```
# save the old solution for comparison with the result of this
# source iteration step.
for imom in range(qdata.nmom):
    for i in xrange(meshdata.ncells):
        for j in xrange(meshdata.nvrtx):
            oldmom[imom][ig][i][j] = momdata[imom][ig][i][j]
            momdata[imom][ig][i][j] = 0.

# loop over angles. Angles proceed independently in the
# absence of implicit boundary conditions. A coarse grained
# parallelization can be implemented by assigning individual
# angles to separate processors at this point.
for kk in range(qdata.nang):
    solved = []
    for ic in xrange(meshdata.ncells):
        solved.append(0)
    omega = qdata.qdpt(kk)
    swp   = swpdata.order(kk)
    m2d   = qdata.mom2dis(kk)

    # loop over cells
    for ic in xrange(meshdata.ncells):
        # find next cell to solve
        icell = swp[ic]
        # create BTE for this cell
        xstot = matprop.sigt( ig, meshdata.region(icell) )
        beta  = matprop.sigstp( ig, meshdata.region(icell) )
        # create the source moments. note that the fixed and
        # inscatter sources could be calculated and stored outside the
        # self-scatter loop for increased computational efficiency.
        if (fsds_prob):
            fixed = fsdsdata.fsdsMom(icell,ig)
        else:
            fixed = fsdata.fsrc(icell,ig)
        insct   = in_scatter(icell, ig, meshdata, matprop, oldmom, qdata)
        selfsct = self_scatter(icell, ig, meshdata, matprop, oldmom, qdata)
        # convert from moment trial space to discrete trial space
        for j in range (meshdata.nvrtx):
            qvec[j] = 0.
            for i in range(qdata.nmom):
                qvec[j] = qvec[j] + m2d[i]*(
                    fixed[i][j] + insct[i][j] + selfsct[i][j] )
        # solve BTE for this cell
        b = BteEquation(meshdata, icell, matprop.icp, omega,
                        xstot, beta, kk)
        b.ludecomp()
        # for increased speed at the cost of more memory, the
        # ludecomp could be stored here for use in future iterations.
        b.rhs(meshdata, qvec, soldata, solved,
              qcsd[csdptr_gm1], psiE)
        b.fbsub()
        solved[icell] = 1
        # store the solution
        soldata[icell] = b.bvec[:]
        if (matprop.icp):
            psiE[icell] = b.psiE
            qcsd[csdptr_g][kk][icell] = b.qcsd[:]
        # return for next cell in this angle.

    # accumulate the moments of the angular solution for
    # computation of scattering source and edits.
    for m in range(qdata.nmom):
        x = qdata.dis2mom(kk)
        for j in range(meshdata.ncells):
            for i in range(meshdata.nvrtx):
                momdata[m][ig][j][i] = momdata[m][ig][j][i] + \
                                       soldata[j][i]*x[m]
```

```
            # accumulate leakage for the balance table edit
            for bf in meshdata.bdry_face:
                xx = 0.; icell = bf[0]; iface = bf[1]
                yy = meshdata.avect(icell,iface)
                for j in range(meshdata.ndim):
                    xx = xx + omega[j]*yy[j]
                if (xx > meshdata.dplimit):
                    x0 = meshdata.face_node[iface][0]
                    x1 = meshdata.face_node[iface][1]
                    x2 = meshdata.face_node[iface][2]
                    leakage[ig] = leakage[ig] + \
                                  (1./3.)*omega[meshdata.ndim]*xx*( \
                          soldata[icell][x0] + \
                          soldata[icell][x1] + \
                          soldata[icell][x2] )

            # accumulate the CSD energy source and removal terms for
            # the balance table.
            if (matprop.icp):
                wgt = omega[meshdata.ndim]; x0 = 0.; x1 = 0.
                for j in range(meshdata.ncells):
                    for i in range(meshdata.nvrtx):
                        x0 = x0 + qcsd[csdptr_gm1][kk][j][i]
                        x1 = x1 + qcsd[csdptr_g][kk][j][i]
                csd_src[ig] = csd_src[ig] + wgt*x0
                csd_rem[ig] = csd_rem[ig] + wgt*x1

            # return for next angle in this energy group

        # Compute a measure of the change in the solution
        delta = 0.
        for i in xrange(meshdata.ncells):
            for j in xrange(meshdata.nvrtx):
                delta = delta + (oldmom[0][ig][i][j] - momdata[0][ig][i][j])**2
        print " ", icount, "delta= ", delta

        # At this point convergence of the scattering source can be
        # accelerated with any number of pre-conditioning techniques
        # such as diffusion synthetic acceleration (DSA). This is
        # essential for problems with a high within groups scattering
        # ratio.

        # return for next within-group scattering source iteration.

    print "Group ", ig, "converged."
    print

    # Compute other balance table entries
    imom = 0
    for j in range(meshdata.ncells):
        if (fsds_prob):
            xx = fsdsdata.fsdsMom(j,ig)[0]
        else:
            xx = fsdata.fsrc(j,ig)[0]
        yy = in_scatter(j, ig, meshdata, matprop, momdata, qdata)[0]
        for i in range(meshdata.nvrtx):
            abs[ig] = abs[ig] + momdata[imom][ig][j][i]* ( \
                      matprop.sigt( ig, meshdata.region(j)) - \
                      matprop.sigsct(0,ig,ig,meshdata.region(j)) )* \
                      meshdata.vol(j)
            source[ig] = source[ig] + meshdata.vol(j)*( xx[i] + yy[i] )
    abs[ig] = 0.25*abs[ig] + csd_rem[ig]
    source[ig] = 0.25*source[ig] + csd_src[ig]

    # return for next energy group

# At the end of the loop over energy groups, if up-scatter is present,
# diffusion synthetic or other acceleration techniques similar to
```

```
# those employed for the acceleration of the self scattering source
# can be employed to speed convergence at this point. Since this demo
# is restricted to down-scatter for purposes of simplicity,
# no outer loop acceleration algorithms are necessary.

# If this is a first scattered distributed source problem, then we
# must add in the un-collided solution component to the collided
# component to complete the solution.
if (fsds_prob):
    fsdsdata.total_solution(momdata)

# Solution is complete at this point. Now we begin post-processing.
# If the solution is saved to disk here, then any desired post processing
# subsequent to this run can be accomplished without the work of
# re-computing the solution.

# Write a visualization link file.
gmvlink(meshdata, momdata, "gmv.out")

# Compute an example dose edit by integrating the results over
# the finite element mesh. Other edits can be computed in a
# similar manner as required. If edits at an arbitrary point are
# required this is also easily computed as the finite element trial
# space provides a rigorous means of interpolation at any point
# in the mesh.
for ig in range(matprop.ngroups):
    result = 0.
    imom = 0
    for j in range(meshdata.ncells):
        for i in range(meshdata.nvrtx):
            result = result + momdata[imom][ig][j][i]*  \
                     matprop.sigreac( ig, meshdata.region(j) )* \
                     meshdata.vol(j)
result = 0.25*result
print "Dose for entire mesh: ", result
print

# Compute an example result at an arbitrary point in the mesh. Call
# the TetTrace interpolation routines to produce the correct LDFEM
# value for group 0, moment 0 at point (0.5, 0.6, 0.7). Also find the
# cell number containing this point for reference.
pt = (0.5, 0.6, 0.7)
trace = TetTrace(meshdata)
x = trace.fieldVal( pt, momdata[0][0] )
i = trace.cellNumb( pt )
print "Results in group 0, moment 0, (x=0.5, y=0.6, z=0.7)"
print "  phi0= ", x, ", in cell number: ", i
print

# Complete a balance table and print. The balance table can be used
# as an error checking tool, a convergence metric, or as
# another set of edit quantities of interest to the user.
print "Balance Table--"
print "  leakage        = ", leakage
print "  other removal  = ", abs
print "  total source   = ", source
for ig in range(matprop.ngroups):
    if (source[ig] != 0.):
        balance[ig] = 1. - (abs[ig]+leakage[ig])/source[ig]
    else:
        balance[ig] = abs[ig]+leakage[ig]
print "  balance        = ", balance

# Post processing complete.

# Write a footer and halt.
print "\nTransport Solution Complete.\n\n"
```

```
# ------------------------------------------------------------------
#                        end of frost.py
# ------------------------------------------------------------------

#-------------------------------------------------------------------
#
# File: fsds.py
#
#      Provides information related to the
#      "First-Scattered-Distributed-Source" solution technique.
#
#      Computes the  un-collided component of a transport solution
#      using an analytic ray tracing technique, then forms a first
#      scattered distributed source.
#
#      This simple example assumes a single isotropic point source.
#      A general purpose routine would provide for
#      the entry of any number of user specified multi-group
#      anisotropic sources.
#
# Author: John McGhee, Radion Technologies
#
# Date  : 19 Feb 2004

#-------------------------------------------------------------------
#
# $Id: fsds.py,v 1.9 2004/03/10 17:17:26 mcghee Exp $
#
#-------------------------------------------------------------------

from sys import exit

class Fsds:
    "First Scattered Distributed Source methods and data"

    def __init__(self, matprop, meshdata, qdata, fsdata):
        "Computes uncollided solution component and first scattered source"
        from math import sqrt, pi, exp
        from sph import sphfun
        from trace import TetTrace

        self.ngroups  = matprop.ngroups
        self.nvrtx    = meshdata.nvrtx
        self.nmom     = qdata.nmom
        self.ncells   = meshdata.ncells
        self.ncells_per_ray = 0
        self.nrays    = 0
        self.uncolMom = []   #[icell][ig][imom][ivrtx]
        self.sctsrc   = []   #[icell][ig][imom][ivrtx]

        # A production algorithm would likely provide for multiple
        # sources by including a loop over all the sources in the
        # problem at this point. As we assume only one source for this
        # demo, no loop over sources is required.

        # Get the point source location and strength. An isotropic
        # source is assumed herein for simplicity, but in general this
        # is not a restriction. In a production code, the source
        # details would be specified by the user at problem setup.
        srcID = 0
        (ray_starting_point, ptSource) = fsdata.isopsrc(srcID)

        # To form the uncollided solution, we track
        # from the source to the Gaussian integration point(s) on each
        # cell. The first scattered source is then rigorously computed
        # by using finite element integration rules on the cell. Here
        # we go.
```

```
            # As an example we use a four point quadratic Gaussian
            # integration rule on linear tetrahedra. Other integration
            # rules could be used if desired.
            self.nqdpt = 4
            alpha = 0.05*(5. + 3.*sqrt(5.) )
            beta  = 0.05*(5. -    sqrt(5.) )
            gamma = 4.*alpha - 3.*beta
            delta = 4.*beta  - (alpha + 2.*beta)

            # Loop over cells computing the un-collided solution component and
            # the first scattered source.
            ncpr = 0
            for icell in xrange(self.ncells):
                n0 = meshdata.cell_node[icell][0]
                n1 = meshdata.cell_node[icell][1]
                n2 = meshdata.cell_node[icell][2]
                n3 = meshdata.cell_node[icell][3]
                c0 = meshdata.node_coord[n0]
                c1 = meshdata.node_coord[n1]
                c2 = meshdata.node_coord[n2]
                c3 = meshdata.node_coord[n3]
                phi= []   #[iqdpt][ig][imom]
                # Loop over quadrature points
                for iqdpt in range(self.nqdpt):
                    # Compute the coordinates of the Gaussian integration point
                    if (iqdpt == 0):
                        ray_ending_point = (
                            alpha*c0[0] + beta*(c1[0]+c2[0]+c3[0]),
                            alpha*c0[1] + beta*(c1[1]+c2[1]+c3[1]),
                            alpha*c0[2] + beta*(c1[2]+c2[2]+c3[2]) )
                    elif (iqdpt == 1):
                        ray_ending_point = (
                            alpha*c1[0] + beta*(c0[0]+c2[0]+c3[0]),
                            alpha*c1[1] + beta*(c0[1]+c2[1]+c3[1]),
                            alpha*c1[2] + beta*(c0[2]+c2[2]+c3[2]) )
                    elif (iqdpt == 2):
                        ray_ending_point = (
                            alpha*c2[0] + beta*(c0[0]+c1[0]+c3[0]),
                            alpha*c2[1] + beta*(c0[1]+c1[1]+c3[1]),
                            alpha*c2[2] + beta*(c0[2]+c1[2]+c3[2]) )
                    else:
                        ray_ending_point = (
                            alpha*c3[0] + beta*(c0[0]+c1[0]+c2[0]),
                            alpha*c3[1] + beta*(c0[1]+c1[1]+c2[1]),
                            alpha*c3[2] + beta*(c0[2]+c1[2]+c2[2]) )

                    # Compute the direction of particle travel.
                    omega = ( ray_ending_point[0] - ray_starting_point[0],
                              ray_ending_point[1] - ray_starting_point[1],
                              ray_ending_point[2] - ray_starting_point[2] )
                    # At this point the algorithm tracks through
                    # the mesh tet by tet from the starting point to the
                    # ending point of the ray, accumulating the decay of the source
                    # based on the material assigned to each tet and the path
                    # length in that tet. Alternatively, the ray could be
                    # traced through the elements of any other problem
                    # related geometry deemed appropriate, for
                    # example a voxel based CT scan. The only output
                    # required from the tracking algorithm is the optical
                    # path length from source to  quadrature point. Methods
                    # provided in trace.py (TetTrace class) provide an
                    # example of how this can be accomplished on tet meshes.
                    trace = TetTrace(meshdata)
                    ncpr = ncpr + trace.trackData(icell, ray_ending_point,
ray_starting_point)
                    # Compute the path length and normalization constant
                    path  = sqrt(omega[0]**2 + omega[1]**2 + omega[2]**2)
                    xnorm = 1./(4.*pi*path*path)
```

```
                # Loop over energy groups
                yy = [] #[ngroups][nmom]
                for ig in range(matprop.ngroups):
                    # compute the optical path
                    optical_path = 0.
                    for i in trace.path:
                        imat = meshdata.cell_region[i[0]]
                        optical_path = optical_path + i[1]*matprop.sigt(ig,imat)
                    # compute the uncollided angular flux at this point
                    psi = xnorm*ptSource[ig]*exp(-optical_path)
                    # compute the moments of the uncollided flux
                    # at this quadrature point.
                    xx = [] #[nmom]
                    for imom in range(qdata.nmom):
                        l = qdata.index_lm[imom][0]
                        m = qdata.index_lm[imom][1]
                        xx.append( psi*sphfun(l, m, omega[0], omega[1], omega[2]) )
                    yy.append(xx)
                phi.append(yy) #[nqdpt][ngroups][nmom]

                # return for the next quadrature point in this cell

            # Now that we have the uncollided solution moments at all
            # the quadrature points in this cell, we map the solution
            # to the cell vertexes. This is done via a rigorous finite
            # element based formula that preserves as many spatial
            # moments of the solution as possible. This
            # amounts to assigning a specific weighted sum of the solution
            # at the quadrature points to the cell vertexes.
            zz = [] #[ngroups][nmom][nvrtx]
            for ig in range(matprop.ngroups):
                yy = [] #[nmom][nvrtx]
                for imom in range(qdata.nmom):
                    xx = [] #[nvrtx]
                    xx.append( gamma*phi[0][ig][imom] + delta*(
                        phi[1][ig][imom] + phi[2][ig][imom] + phi[3][ig][imom]))
                    xx.append( gamma*phi[1][ig][imom] + delta*(
                        phi[0][ig][imom] + phi[2][ig][imom] + phi[3][ig][imom] ))
                    xx.append( gamma*phi[2][ig][imom] + delta*(
                        phi[1][ig][imom] + phi[0][ig][imom] + phi[3][ig][imom] ))
                    xx.append( gamma*phi[3][ig][imom] + delta*(
                        phi[1][ig][imom] + phi[2][ig][imom] + phi[0][ig][imom] ))
                    yy.append(xx)
                zz.append(yy)
            self.uncolMom.append(zz) #[ncells][ngroups][nmom][nvrtx]

            # Now that we have the moments of the uncollided solution
            # component on this cell, we can compute the moments of the
            # first scattered source (particles/time-length^3).
            imat = meshdata.region(icell)
            zz = [] #[ngroups][nmom][nvrtx]
            for isink in range(matprop.ngroups):
                yy = [] #[nmom][nvrtx]
                for imom in range(qdata.nmom):
                    l = qdata.index_lm[imom][0]
                    xx = [] #[nvrtx]
                    for ivrtx in range(meshdata.nvrtx):
                        x = 0.
                        for isrc in range(matprop.ngroups):
                            x = x + self.uncolMom[icell][isrc][imom][ivrtx]* \
                                matprop.sigsct(l, isrc, isink, imat)
                        xx.append(x)
                    yy.append(xx)
                zz.append(yy)
            self.sctsrc.append(zz) #[ncells][ngroups][nmom][nvrtx]

        self.nrays = self.nqdpt*self.ncells
        self.ncells_per_ray = float(ncpr)/float(self.nrays)
```

```
    # ------------------------------------------------------------------------

    def fsdsMom(self, icell, igroup):
        "Returns the first scattered distributed source for cell icell, and group
igroup"
        if (icell < 0) or (icell >= self.ncells):
            print "Error: icell argument out of range in fsds"
            exit()
        if (igroup < 0) or (igroup >= self.ngroups):
            print "Error: igroup argument out of range in fsds"
            exit()
        return self.sctsrc[icell][igroup] #[nmom][nvrtx]

    # ------------------------------------------------------------------------

    def uncolMom(self,icell,igroup):
        "Returns the moments of the un-collided component of the solution."
        if (icell < 0) or (icell >= self.ncells):
            print "Error: icell argument out of range in fsds"
            exit()
        if (igroup < 0) or (igroup >= self.ngroups):
            print "Error: igroup argument out of range in fsds"
            exit()
        return self.uncolMom[icell][igroup] #[nmom][nvrtx]

    # ------------------------------------------------------------------------

    def total_solution(self, collided):

        "Combines the uncollided and collided solution components to form the total
solution."
        # collided[nmom][ngroups][ncells][nvrtx]
        for imom in range(self.nmom):
            for ig in range(self.ngroups):
                for icell in xrange(self.ncells):
                    for ivrtx in range(self.nvrtx):
                        collided[imom][ig][icell][ivrtx] = \
                        self.uncolMom[icell][ig][imom][ivrtx] + \
                        collided[imom][ig][icell][ivrtx]


#-------------------------------------------------------------------
#                       end of fsds.py
#-------------------------------------------------------------------
#-------------------------------------------------------------------
#
# File: fsrc.py
#
#       Provides information on a fixed source. This source is defined
#       as part of the problem setup. For simplification, this demo
#       provides a single fixed source definition as a Python function
#       call.
#
#       A production code would ordinarily provide a means for the
#       user to define the details of the source(s) as part of the problem
#       setup. This could be done in a variety of ways such as reading
#       the source definition from disk or selecting the source
#       definition from a library of pre-defined functions.
#
# Author: John McGhee, Radion Technologies
#
# Date  : 19 Feb 2004
#-------------------------------------------------------------------
#
# $Id: fsrc.py,v 1.5 2004/03/05 21:51:31 mcghee Exp $
#
```

```
#-----------------------------------------------------------------

from sys import exit

class FixedSource:
    "Provides information on the fixed source."

    def __init__(self, qdata, meshdata, matprop ):
        "Initializes the fixed source class"

        self.nmom     = qdata.nmom
        self.index_lm = qdata.index_lm
        self.nvrtx    = meshdata.nvrtx
        self.ncells   = meshdata.ncells
        self.ngroups  = matprop.ngroups
        self.one = []
        self.zero = []

        for i in range(self.nvrtx):
            self.one.append(1.)
            self.zero.append(0.)

        self.ptSrcLoc = (0., 0., 0.)
        self.ptSrcVal = [1.,]
        for i in range(1,self.ngroups):
            self.ptSrcVal.append(0.)

    # ------------------------------------------------------------------------

    def fsrc(self, icell, igroup):
        "Returns the fixed source for cell icell, and group igroup"

        # This method returns a uniformly distributed, isotropic
        # source with a strength of 1.0 in group 0.

        if (icell < 0) or (icell >= self.ncells):
            print "Error: icell argument out of range in fsrc"
            exit()
        if (igroup < 0) or (igroup >= self.ngroups):
            print "Error: igroup argument out of range in fsrc"
            exit()
        x = []
        for imom in range(self.nmom):
            l = self.index_lm[imom][0]
            if ( l == 0 ) and ( igroup == 0 ):
                x.append( self.one )
            else:
                x.append( self.zero )
        return x #[nmom][nvrtx]

    def isopsrc(self, srcID):
        "Returns the location and strength of an isotropic point source"

        # This method returns a isotropic source located at (0,0,0)
        # with a strength of 1.0 in group 0.

        return ( self.ptSrcLoc, self.ptSrcVal )

#-----------------------------------------------------------------
#                       end of fsrc.py
#-----------------------------------------------------------------
#-----------------------------------------------------------------
#
# File: geom.py
#
#       Mesh geometry and related functions.
#
#       Reads in mesh data from disk: ncells, nvertexes, nregions,
```

```
#       vertex coordinates, cell vertexes,
#       region numbers, boundary conditions, cell types.

#       For simplicity, this demo assumes a 3D linear tetrahdedral
#       mesh, but other geometries can be treated in a similar
#       manner. The finite element geometric coefficients provided
#       herein are normally computed by a
#       spatial quadrature integration rule. For linear
#       tetrahedra, it so happens that the results of these
#       integrations are conviniently expressed in terms of the
#       tet volumes, face areas, and face normal vectors. Other
#       geometries would commonly provide this data as a matrix of
#       coefficients particular to each cell.

#       As a matter of computational efficiency, these "geoemtric"
#       coefficients are pre-computed and stored for each cell. This
#       data is then later combined with the Sn angular quadrature data
#       and material properties data to compute coefficients specific
#       to each space-angle-energy phase space cell.

#       Finite element geomtry data computed for tets:
#          - cell volumes
#          - cell face areas
#          - outward directed cell face unit normal vectors

#       We also compute pointers to adjacent cells and shared vertexes
#       to complete the geometry description.
#
# Author: John McGhee, Radion Technologies
#
# Date  : 19 Feb 2004

#-----------------------------------------------------------------------
#
# $Id: geom.py,v 1.18 2004/03/05 21:51:31 mcghee Exp $
#
#-----------------------------------------------------------------------

from sys import exit

class Geometry:
    "Provides information on the tetrahedral mesh geometry."

    element_geometry = "tetrahedra"
    null_flag = (-999,-999, -999)
    bdry_flag = (-998,-998, -998)
    nvrtx  = 4
    nfaces = 4
    ndim = 3
    face_node = ( (1,2,3), (0,3,2), (0,1,3), (0,2,1) )

    def __init__(self,mesh_file):
        "Reads in the tet mesh geometry from disk and computes volumes, areas,
etc."

        from string import split, atof, atoi, strip
        from math import sqrt

        # Define a constant for test
        self.dplimit = 1.e-15

        # Setup for read of geometry data from disk.
        self.name=mesh_file
        f = open(mesh_file)
        record = f.readline()

        # Read in the node coordinates from disk.
        while (record and strip(record) != "nodes" ):
```

```
            record = f.readline()
        if (record == ""):
            print "Error: unable to find nodes keyword on mesh file %s\n" %
mesh_file
            exit()
        self.node_coord = []
        record = f.readline()
        while (record and strip(record) != "end_nodes"):
            j = split(record)
            self.node_coord.append(map(atof,j[1:4]))
            record = f.readline()
        if (record == ""):
            print "Error: unable to find end_nodes keyword on mesh file %s\n" %
mesh_file
            exit()
        self.nnodes = len(self.node_coord)

        # Read in the cell nodes and region flag from disk.
        # Note the tet node numbering scheme assumes that local node numbers
        # increase in a clockwise direction when viewed from node 0.
        # Also note that the global node numbering scheme is assumed
        # to be zero based, ie the first node in the node list is node
        # "0". Mesh regions are assumed to be id'd with consecutive
        # integers beginning with 0.
        record = f.readline()
        while (record and strip(record) != "cells"):
            record = f.readline()
        if (record == ""):
            print "Error: unable to find cells keyword on mesh file %s\n" %
mesh_file
            exit()
        self.cell_node = []
        self.cell_region = []
        record = f.readline()
        while (record and strip(record) != "end_cells" ):
            j = split(record)
            if (atoi(j[1]) != 4):
                print "Error: Non-tetrahedral cell id %s found on mesh file %s\n" %
(j[0], mesh_file)
                exit()
            self.cell_node.append(map(atoi,j[2:self.nvrtx+2]))
            self.cell_region.append( atoi(j[self.nvrtx+2]) )
            record = f.readline()
        if (record == ""):
            print "Error: unable to find end_cells keyword on mesh file %s\n" %
mesh_file
            exit()
        self.ncells = len(self.cell_node)
        self.nregions = max(self.cell_region)+1

        # Report successful read of mesh geometry from disk.
        f.close()
        print "Mesh data successfully read from file %s" % mesh_file
        print "  ncells     = ", self.ncells
        print "  nnodes     = ", self.nnodes

        # Compute outward directed cell face area vectors by vector cross
        # product of cell edge vectors. Compute cell volume by triple scalar
        # product.
        self.cell_volume = []
        self.area_vector = []
        self.face_area = []
        for nlist in self.cell_node:
            n0 = nlist[0]
            n1 = nlist[1]
            n2 = nlist[2]
            n3 = nlist[3]
            x0 = self.node_coord[n0][0]
```

```
            y0 = self.node_coord[n0][1]
            z0 = self.node_coord[n0][2]
            x1 = self.node_coord[n1][0]
            y1 = self.node_coord[n1][1]
            z1 = self.node_coord[n1][2]
            x2 = self.node_coord[n2][0]
            y2 = self.node_coord[n2][1]
            z2 = self.node_coord[n2][2]
            x3 = self.node_coord[n3][0]
            y3 = self.node_coord[n3][1]
            z3 = self.node_coord[n3][2]
            # face 0
            xx = (
                (  0.5*( (y2-y1)*(z3-z1) - (z2-z1)*(y3-y1) ) ),
                ( -0.5*( (x2-x1)*(z3-z1) - (z2-z1)*(x3-x1) ) ),
                (  0.5*( (x2-x1)*(y3-y1) - (y2-y1)*(x3-x1) ) ) )
            self.area_vector.append(xx)
            self.face_area.append( sqrt(xx[0]*xx[0] + xx[1]*xx[1] +
                                        xx[2]*xx[2]) )
            # face 1
            xx = (
                (  0.5*( (y3-y0)*(z2-z0) - (z3-z0)*(y2-y0) ) ),
                ( -0.5*( (x3-x0)*(z2-z0) - (z3-z0)*(x2-x0) ) ),
                (  0.5*( (x3-x0)*(y2-y0) - (y3-y0)*(x2-x0) ) ) )
            self.area_vector.append(xx)
            self.face_area.append( sqrt(xx[0]*xx[0] + xx[1]*xx[1] +
                                        xx[2]*xx[2]) )
            # face 2
            xx = (
                (  0.5*( (y1-y0)*(z3-z0) - (z1-z0)*(y3-y0) ) ),
                ( -0.5*( (x1-x0)*(z3-z0) - (z1-z0)*(x3-x0) ) ),
                (  0.5*( (x1-x0)*(y3-y0) - (y1-y0)*(x3-x0) ) ) )
            self.area_vector.append(xx)
            self.face_area.append( sqrt(xx[0]*xx[0] + xx[1]*xx[1] +
                                        xx[2]*xx[2]) )
            # face 3
            xx = (
                (  0.5*( (y2-y0)*(z1-z0) - (z2-z0)*(y1-y0) ) ),
                ( -0.5*( (x2-x0)*(z1-z0) - (z2-z0)*(x1-x0) ) ),
                (  0.5*( (x2-x0)*(y1-y0) - (y2-y0)*(x1-x0) ) ) )
            self.area_vector.append(xx)
            self.face_area.append( sqrt(xx[0]*xx[0] + xx[1]*xx[1] +
                                        xx[2]*xx[2]) )
            # cell volume
            self.cell_volume.append( (-1./3.)*( (x3-x0)*xx[0] +
                                                (y3-y0)*xx[1] +
                                                (z3-z0)*xx[2] ) )

        # For cell i, vertex j, face k, create a pointer to the adjacent
        # cell ia and vertex ja. We construct a list of bins so we don't
        # have to search the whole mesh. Each bin contains a list of
        # the cells that share a particular node.
        self.adjacent = []
        bin = []
        for i in xrange( self.nnodes ):
            bin.append([])
        i = -1
        for nlist in self.cell_node:
            i = i+1
            j = -1
            for k in nlist:
                j = j+1
                bin[k].append( (i,j) )
        for icell in xrange(self.ncells):
            for ivrtx in xrange(self.nvrtx):
                for iface in xrange(self.nfaces):
                    if ( iface == ivrtx ):
                        xx = self.null_flag
```

```
                    else:
                        found = 0
                        node = ( self.cell_node[icell][self.face_node[iface][0]],
                                 self.cell_node[icell][self.face_node[iface][1]],
                                 self.cell_node[icell][self.face_node[iface][2]] )
                        for i in bin[self.cell_node[icell][ivrtx]]:
                            if (i[0] == icell):
                                continue
                            j = self.cell_node[i[0]]
                            if ( (node[0] in j) and (node[1] in j)
                                 and (node[2] in j) ):
                                if ( not ( j[0] in node ) ):
                                    jface = 0
                                elif  ( not ( j[1] in node ) ):
                                    jface = 1
                                elif  ( not ( j[2] in node ) ):
                                    jface = 2
                                else:
                                    jface = 3
                                found = 1
                                xx = (i[0], i[1], jface)
                                break
                        if (found != 1):
                            xx = self.bdry_flag
                    self.adjacent.append(xx)

        # Make a list of the boundary faces
        self.bdry_face = []
        for icell in xrange(self.ncells):
             for iface in xrange(self.nfaces):
                 if (self.adjcell(icell, iface)[0] == self.bdry_flag[0]):
                     self.bdry_face.append( (icell, iface) )
        self.nbdry_faces = len(self.bdry_face)
        print "  nbdry_faces= ", self.nbdry_faces

        # Check the calculations, and report success (or failure!).
        print "  Calculation of mesh geometry data complete, checking results...."
        if ( self.invariant() ):
            print "  Mesh data checks passed.\n"
        else:
            print "  Mesh data checks failed!\n"
            exit()

    # ------------------------------------------------------------------------

    def vol(self,i):
        "Returns the volume of cell i"
        if (i < 0) or (i >= self.ncells):
            print "Error: argument out of range in geometry.vol"
            exit()
        return self.cell_volume[i]

    # ------------------------------------------------------------------------

    def area(self,i,j):
        "Returns the area in cell i of face j"
        from sys import exit
        if (i < 0) or (i >= self.ncells):
            print "Error: argument i out of range in geometry.area"
            exit()
        if (j < 0) or (j >= self.nfaces):
            print "Error: argument j out of range in geometry.area"
            exit()
        ii = i*self.nfaces + j
        return self.face_area[ii]

    # ------------------------------------------------------------------------
```

```
    def region(self,i):
        "Returns the mesh region number of cell i"
        if (i < 0) or (i >= self.ncells):
            print "Error: argument out of range in geometry.region"
            exit()
        return self.cell_region[i]

    # --------------------------------------------------------------------

    def avect(self,i,j):
        "Returns the face area vector in cell i of face j"
        if (i < 0) or (i >= self.ncells):
            print "Error: argument i out of range in geometry.avect"
            exit()
        if (j < 0) or (j >= self.nfaces):
            print "Error: argument j out of range in geometry.avect"
            exit()
        ii = i*self.nfaces + j
        return self.area_vector[ii]

    # --------------------------------------------------------------------

    def adjvrtx(self,i,j,k):
        "Returns the adjacent cell and vertex associated with cell i, vertex j,
face k"
        if (i < 0) or (i >= self.ncells):
            print "Error: argument i out of range in geometry.adjvrtx"
            exit()
        if (j < 0) or (j >= self.nvrtx):
            print "Error: argument j out of range in geometry.adjvrtx"
            exit()
        if (k < 0) or (k >= self.nfaces):
            print "Error: argument k out of range in geometry.adjvrtx"
            exit()
        ii = i*self.nfaces*self.nvrtx + j*self.nvrtx + k
        return self.adjacent[ii]

    # --------------------------------------------------------------------

    def adjcell(self,i,k):
        "Returns the adjacent cell and face associated with cell i, face k"
        if (i < 0) or (i >= self.ncells):
            print "Error: argument i out of range in geometry.adjcell"
            exit()
        if (k < 0) or (k >= self.nfaces):
            print "Error: argument k out of range in geometry.adjcell"
            exit()
        j = self.face_node[k][0]
        ii = i*self.nfaces*self.nvrtx + j*self.nvrtx + k
        return ( self.adjacent[ii][0], self.adjacent[ii][2] )

    # --------------------------------------------------------------------

    def invariant(self):
        "Checks the mesh data for internal consistency"
        if (len(self.cell_region) != self.ncells):
            print "Error: region data size != ncells"
            exit()
        if (len(self.cell_node) != self.ncells):
            print "Error: cell_node data size != ncells"
            exit()
        if (len(self.cell_volume) != self.ncells):
            print "Error: cell volume data size != ncells"
            exit()
        if (len(self.node_coord) != self.nnodes):
            print "Error: node_coord data size != nnodes"
            exit()
        if (len(self.face_area) != self.ncells*self.nfaces):
```

```
            print "Error: face area data size != ncells*nfaces"
            exit()
        if (len(self.area_vector) != self.ncells*self.nfaces):
            print "Error: face area vector data size != ncells*nfaces"
            exit()
        if (len(self.node_coord) != self.nnodes):
            print "Error: node_coord data size != nnodes"
            exit()
        if (len(self.adjacent) != self.ncells*self.nvrtx*self.nfaces):
            print "Error: adjacent pointer data size incorrect!"
            exit()
        if ( min(self.cell_volume) <= 0. ):
            print "Error: Zero or negative cell volumes found."
            exit()
        if ( min(self.face_area) <=0. ):
            print "Error: Zero or negative cell face areas found."
            exit()
        for icell in xrange(self.ncells):
            for ivrtx in xrange(self.nvrtx):
                for iface in xrange(self.nfaces):
                    xx = self.adjvrtx(icell,ivrtx,iface)
                    jcell = xx[0]
                    jvrtx = xx[1]
                    if (jcell < 0):
                        continue
                    else:
                        found = 0
                        for i in xrange(self.nfaces):
                            yy = self.adjvrtx(jcell,jvrtx,i)
                            if (yy[0] == icell) and (yy[1] == ivrtx):
                                jface = i
                                found = 1
                                break
                        if ( found != 1 ):
                            print "Error: no matching pointer in adjacent array."
                            exit()
                        if ( self.cell_node[jcell][jvrtx] !=
                             self.cell_node[icell][ivrtx] ):
                            print "Error: cell node mis-match in adjacent array."
                            exit()
                        if ( self.adjcell(icell,iface) != ( jcell, jface ) ):
                            print "Error: adjacent cell pointers do not match"
                            exit()
                        if ( self.adjcell(jcell,jface) != ( icell, iface  ) ):
                            print "Error: adjacent cell pointers do not match"
                            exit()
                        for i in xrange(3):
                            zz = self.avect(icell,iface)[i] + \
                                 self.avect(jcell,jface)[i]
                            if ( abs(zz) > 1.e-15):
                                print "Error found in face area vectors"
                                exit()
        return 1

#-----------------------------------------------------------------------
#                       end of geom.py
#-----------------------------------------------------------------------
#-----------------------------------------------------------------------
#
# File: matp.py
#
#       Provides multi-group material properties
#       information. Anisotropic up, down, and self scatter
#       are provided for. Anisotropic scattering
#       data is provided in terms of Legendre (Pn) moments of
#       the scattering cross sections.
#
#       All groups and materials are assumed to have the same
```

```
#       Pn scattering order for this demo code. However, considerable
#       memory and computational expense may be saved in a production
#       code by allowing the Pn scattering order to vary from one
#       group to another within a material. Likewise, each material
#       could be allowed to have its own individual Pn order.
#
# Author: John McGhee, Radion Technologies
#
# Date  : 19 Feb 2004

#-----------------------------------------------------------------------
#
# $Id: matp.py,v 1.6 2004/03/05 21:51:31 mcghee Exp $
#
#-----------------------------------------------------------------------

from sys import exit

class MaterialProps:
    "Provides material property data."

    def __init__(self, matprop_file):
        "Reads material properties data from disk."

        from string import split, atof, atoi, strip

        # Setup for read of fixed matprop data from disk.
        self.name=matprop_file
        f = open(matprop_file)
        record = f.readline()

        # Read dimensions of the material properties from disk.
        self.matprop = []
        while (record and strip(record) != "dimensions" ):
            record = f.readline()
        if (record == ""):
            print "Error: unable to find dimensions keyword on matprop file %s\n" %
matprop_file
            exit()
        record = f.readline()
        j = split(record)
        if (len(j) != 4):
            print "Error: Incorrect number of dimensions in matprop file"
        self.ngroups=atoi(j[0])
        self.nscxs = atoi(j[1])
        self.nmat = atoi(j[2])
        self.icp = atoi(j[3])
        record = f.readline()
        while (record and strip(record) != "end_dimensions" ):
            record = f.readline()
        if (record == "" ):
            print "Error: unable to find end_dimensions keyword on matprop file
%s\n" % matprop_file
            exit()
        if (self.ngroups <= 0) or (self.nscxs <= 0) or (self.nmat <= 0):
            print "Error: dimensions out of range on matprop file %s\n" %
matprop_file
            exit()

        # Read in the matprop values from disk.
        while (record and strip(record) != "material_data" ):
            record = f.readline()
        if (record == ""):
            print "Error: unable to find matprop keyword on matprop file %s\n" %
matprop_file
            exit()

        self.matprop = []
```

```
            record = f.readline()
            while (record and strip(record) != "end_material_data"):
                j = split(record)
                if (len(j) == 0):
                    print "Error: blank line found in matprop data."
                    exit()
                self.matprop.append(map(atof,j))
                record = f.readline()
            if (record == ""):
                print "Error: unable to find end_matprop keyword on matprop file %s\n"
% matprop_file
                exit()
            self.nmatprop = len(self.matprop)

            ii = (self.nscxs+1)*self.nmat*self.ngroups
            if (self.nmatprop != ii):
                print "Error: incorrect number of matprop values on file"

        # ---------------------------------------------------------------------

        def sigsct(self, l, isrc, isink, imat):
            "Gets scattering cross section."
            if (imat < 0) or (imat >= self.nmat):
                print "Error: imat argument out of range in scat"
                exit()
            if (isink < 0) or (isink >= self.ngroups):
                print "Error: isink argument out of range in scat"
                exit()
            if (isrc < 0) or (isrc >= self.ngroups):
                print "Error: isrc argument out of range in scat"
                exit()
            if (l < 0):
                print "Error: l argument out of range in scat"
                exit()
            if (l >= self.nscxs):
                return 0.
            else:
                ii = imat*self.ngroups*(self.nscxs+1) + isrc*(self.nscxs+1) + 1 + l
                return self.matprop[ii][isink]

         # ---------------------------------------------------------------------

        def sigt(self, igroup, imat):
            "Gets total cross section."
            if (imat < 0) or (imat >= self.nmat):
                print "Error: imat argument out of range in sigt"
                exit()
            if (igroup < 0) or (igroup >= self.ngroups):
                print "Error: igroup argument out of range in sigt"
                exit()
            ii = imat*self.ngroups*(self.nscxs+1) + igroup*(self.nscxs+1)
            return self.matprop[ii][0]

        # ---------------------------------------------------------------------

        def sigreac(self, igroup, imat):
            "Gets reaction cross section."
            if (imat < 0) or (imat >= self.nmat):
                print "Error: imat argument out of range in reac"
                exit()
            if (igroup < 0) or (igroup >= self.ngroups):
                print "Error: igroup argument out of range in reac"
                exit()
            ii = imat*self.ngroups*(self.nscxs+1) + igroup*(self.nscxs+1)
            return self.matprop[ii][1]

        # ---------------------------------------------------------------------
```

```
    def sigstp(self, igroup, imat):
        "Gets the stopping power over delta E."
        if (imat < 0) or (imat >= self.nmat):
            print "Error: imat argument out of range in sigstp"
            exit()
        if (igroup < 0) or (igroup >= self.ngroups):
            print "Error: igroups argument out of range in sigstp"
            exit()
        ii = imat*self.ngroups*(self.nscxs+1) + igroup*(self.nscxs+1)
        return self.matprop[ii][2]

#-----------------------------------------------------------------
#                        end of matp.py
#-----------------------------------------------------------------

# Multi-group Material Properties data
#
# Format:
# units are 1/cm for cross sections,
# MeV for energy. scattering cross sections
# are given in Legendre moments.
#
# dimensions ngroups, nscxs, nmaterials, icp
#
# Material mat=0
# Group g=0
# sigma-t sig-reac sigma-stopping-power/delta-E
# sigma-0,g->1, sigma-0,g->2, sigma-0,g->3.... sigma-0,g->ngroups
# sigma-1,g->1, ......
# .
# .
# sigma-nscxs,g->1
#
# Group g=1
#   etc.
#
# Material mat=1
# etc.

dimensions
  2 2 2 0
end_dimensions

material_data
 2.00 0.50  0.10
 0.50 0.20
 0.05 0.02
 4.00 1.00  0.20
 0.00 0.40
 0.00 0.04
 1.00 0.25 0.05
 0.25 0.10
 0.025 0.01
 2.00 0.50 0.10
 0.00 0.20
 0.00 0.02
end_material_data

# mesh geometry description
# a (1 x 1 x 1) cube centered at (0.5, 0.5, 0.5) with
# 2 regions, 862 tet cells, 207 nodes, and 200 triangular
# boundary faces. Node numbering is zero based. Mesh regions
# are assumed to be id'd with consecutive integers beginning
# with 0.

nodes
    0    0.0000000000E+00    4.0000000000E-01    0.0000000000E+00
    1    0.0000000000E+00    1.0000000000E+00    1.0000000000E+00
```

2	1.0000000000E+00	1.0000000000E+00	1.0000000000E+00
3	1.0000000000E+00	0.0000000000E+00	1.0000000000E+00
4	0.0000000000E+00	0.0000000000E+00	1.0000000000E+00
5	0.0000000000E+00	1.0000000000E+00	0.0000000000E+00
6	1.0000000000E+00	1.0000000000E+00	0.0000000000E+00
7	1.0000000000E+00	0.0000000000E+00	0.0000000000E+00
8	0.0000000000E+00	4.0000000000E-01	4.0000000000E-01
9	4.0000000000E-01	4.0000000000E-01	4.0000000000E-01
10	4.0000000000E-01	0.0000000000E+00	4.0000000000E-01
11	0.0000000000E+00	0.0000000000E+00	4.0000000000E-01
12	4.0000000000E-01	4.0000000000E-01	0.0000000000E+00
13	4.0000000000E-01	0.0000000000E+00	0.0000000000E+00
14	0.0000000000E+00	2.0000000000E-01	0.0000000000E+00
15	0.0000000000E+00	2.0000000000E-01	2.0000000000E-01
16	2.0000000000E-01	2.0000000000E-01	2.0000000000E-01
17	2.0000000000E-01	0.0000000000E+00	2.0000000000E-01
18	0.0000000000E+00	0.0000000000E+00	2.0000000000E-01
19	2.0000000000E-01	2.0000000000E-01	0.0000000000E+00
20	2.0000000000E-01	0.0000000000E+00	0.0000000000E+00
21	0.0000000000E+00	0.0000000000E+00	0.0000000000E+00
22	0.0000000000E+00	0.0000000000E+00	7.0000000000E-01
23	1.0000000000E+00	0.0000000000E+00	6.8750000000E-01
24	1.0000000000E+00	0.0000000000E+00	3.1175142799E-01
25	1.0000000000E+00	0.0000000000E+00	1.9335937500E-01
26	1.0000000000E+00	1.0000000000E+00	6.8750000000E-01
27	1.0000000000E+00	1.0000000000E+00	3.8671875000E-01
28	1.0000000000E+00	1.0000000000E+00	1.9335937500E-01
29	0.0000000000E+00	1.0000000000E+00	6.8750000000E-01
30	0.0000000000E+00	1.0000000000E+00	3.8671875000E-01
31	0.0000000000E+00	1.0000000000E+00	1.9335937500E-01
32	6.8750000000E-01	1.0000000000E+00	1.0000000000E+00
33	3.8671875000E-01	1.0000000000E+00	1.0000000000E+00
34	1.9335937500E-01	1.0000000000E+00	1.0000000000E+00
35	1.0000000000E+00	3.1250000000E-01	1.0000000000E+00
36	1.0000000000E+00	6.1328125000E-01	1.0000000000E+00
37	1.0000000000E+00	8.0664062500E-01	1.0000000000E+00
38	3.1250000000E-01	0.0000000000E+00	1.0000000000E+00
39	5.5389985616E-01	0.0000000000E+00	1.0000000000E+00
40	8.0664062500E-01	0.0000000000E+00	1.0000000000E+00
41	0.0000000000E+00	6.8750000000E-01	1.0000000000E+00
42	0.0000000000E+00	4.1582841730E-01	1.0000000000E+00
43	0.0000000000E+00	1.9335937500E-01	1.0000000000E+00
44	6.8750000000E-01	1.0000000000E+00	0.0000000000E+00
45	3.8671875000E-01	1.0000000000E+00	0.0000000000E+00
46	1.9335937500E-01	1.0000000000E+00	0.0000000000E+00
47	1.0000000000E+00	3.1250000000E-01	0.0000000000E+00
48	1.0000000000E+00	6.1328125000E-01	0.0000000000E+00
49	1.0000000000E+00	8.0664062500E-01	0.0000000000E+00
50	7.0000000000E-01	0.0000000000E+00	0.0000000000E+00
51	0.0000000000E+00	7.0000000000E-01	0.0000000000E+00
52	4.0000000000E-01	0.0000000000E+00	2.0000000000E-01
53	4.0000000000E-01	4.0000000000E-01	2.0000000000E-01
54	0.0000000000E+00	4.0000000000E-01	2.0000000000E-01
55	2.0000000000E-01	4.0000000000E-01	4.0000000000E-01
56	4.0000000000E-01	2.0000000000E-01	4.0000000000E-01
57	2.0000000000E-01	0.0000000000E+00	4.0000000000E-01
58	0.0000000000E+00	2.0000000000E-01	4.0000000000E-01
59	2.0000000000E-01	4.0000000000E-01	0.0000000000E+00
60	4.0000000000E-01	2.0000000000E-01	0.0000000000E+00
61	2.0000000000E-01	2.0000000000E-01	4.0000000000E-01
62	2.0000000000E-01	4.0000000000E-01	2.0000000000E-01
63	4.0000000000E-01	2.0000000000E-01	2.0000000000E-01
64	0.0000000000E+00	1.9816277408E-01	8.5265650289E-01
65	0.0000000000E+00	3.6429795369E-01	6.0022303814E-01
66	0.0000000000E+00	6.9020637049E-01	8.1647869761E-01
67	0.0000000000E+00	6.8143850763E-01	5.0305088449E-01
68	0.0000000000E+00	8.5642346013E-01	2.0469430523E-01

69	0.0000000000E+00	5.4274727097E-01	2.1012134137E-01
70	0.0000000000E+00	1.5658658666E-01	5.4092971210E-01
71	5.4286513258E-01	0.0000000000E+00	1.5566251981E-01
72	1.6114548438E-01	0.0000000000E+00	5.3892617058E-01
73	3.0058947384E-01	0.0000000000E+00	8.1720302964E-01
74	3.8506624883E-01	0.0000000000E+00	5.8953247729E-01
75	7.9513906813E-01	0.0000000000E+00	8.2792057671E-01
76	6.3890496821E-01	0.0000000000E+00	3.8698326115E-01
77	8.7377401275E-01	0.0000000000E+00	2.0445757272E-01
78	8.5520505408E-01	7.9331699083E-01	0.0000000000E+00
79	5.1427246296E-01	6.9535867584E-01	0.0000000000E+00
80	2.0436734847E-01	8.5696128358E-01	0.0000000000E+00
81	2.1056348860E-01	5.4411661671E-01	0.0000000000E+00
82	5.4576935545E-01	2.0873396013E-01	0.0000000000E+00
83	8.1762242372E-01	1.8451589823E-01	0.0000000000E+00
84	7.5109655258E-01	4.9052150087E-01	0.0000000000E+00
85	1.0000000000E+00	8.2402927123E-01	1.7413106030E-01
86	1.0000000000E+00	1.2487433107E-01	2.0700356803E-01
87	1.0000000000E+00	7.9285776851E-01	8.5510723799E-01
88	1.0000000000E+00	2.0144018136E-01	7.9444561904E-01
89	1.0000000000E+00	4.8428942222E-01	2.5701066922E-01
90	1.0000000000E+00	7.1967437916E-01	5.1294207125E-01
91	1.0000000000E+00	2.7808336078E-01	4.7588364265E-01
92	1.0000000000E+00	4.6606445313E-01	7.5459188612E-01
93	1.7294999713E-01	1.0000000000E+00	1.7572284990E-01
94	8.5207743342E-01	1.0000000000E+00	2.0504346565E-01
95	2.0649949350E-01	1.0000000000E+00	8.5711728089E-01
96	8.0865970785E-01	1.0000000000E+00	8.1686297897E-01
97	5.0279034487E-01	1.0000000000E+00	2.4524971658E-01
98	2.5161078888E-01	1.0000000000E+00	5.2225394214E-01
99	6.8115234375E-01	1.0000000000E+00	5.1855468750E-01
100	5.3393554688E-01	1.0000000000E+00	7.5927734375E-01
101	9.3139665146E-02	1.2553485179E-01	1.0000000000E+00
102	1.8632077853E-01	2.6437242858E-01	1.0000000000E+00
103	2.0524985322E-01	8.5230823230E-01	1.0000000000E+00
104	8.0539233807E-01	1.4690864190E-01	1.0000000000E+00
105	8.5732329126E-01	7.9149776183E-01	1.0000000000E+00
106	2.3515465154E-01	5.2583210433E-01	1.0000000000E+00
107	4.8698290591E-01	2.5022079020E-01	1.0000000000E+00
108	7.6836884278E-01	4.4525994821E-01	1.0000000000E+00
109	5.3945232577E-01	6.6623636466E-01	1.0000000000E+00
110	5.3129423908E-01	3.6301742777E-01	1.5615253846E-01
111	5.2142553338E-01	1.9223792132E-01	1.6863581000E-01
112	6.9057450500E-01	1.7024114100E-01	1.9396915061E-01
113	6.3589775149E-01	3.3465936049E-01	1.5092855295E-01
114	5.2610324720E-01	3.5006957730E-01	3.2026464946E-01
115	5.3892400556E-01	1.7231611860E-01	3.2929723165E-01
116	6.6999320000E-01	1.5999830000E-01	3.2999660000E-01
117	6.4736087761E-01	3.0573460412E-01	3.2217274456E-01
118	1.6280610078E-01	7.3718449692E-01	1.6134962498E-01
119	1.5568563329E-01	5.5970950946E-01	1.5509874015E-01
120	3.3071891700E-01	5.5884902893E-01	1.6928060623E-01
121	2.8757807380E-01	7.0027992714E-01	1.5926615033E-01
122	1.6153279295E-01	6.9803521734E-01	2.8470005310E-01
123	1.7051505099E-01	5.5711183509E-01	3.2806298508E-01
124	3.4376014959E-01	5.4232782442E-01	3.4041747705E-01
125	2.9479372820E-01	6.6063039019E-01	2.9101613137E-01
126	5.1698251895E-01	5.5944707707E-01	1.8200762808E-01
127	6.7969530916E-01	5.2274344087E-01	1.8362101945E-01
128	5.1618508017E-01	5.2586645378E-01	3.3424519004E-01
129	6.8977402522E-01	4.7535559560E-01	3.0938541431E-01
130	4.5891086203E-01	7.4247179192E-01	2.0331723849E-01
131	4.6699594988E-01	6.8829280667E-01	2.9446984286E-01
132	6.6482681022E-01	7.0275413419E-01	1.7337280910E-01
133	6.5235353985E-01	6.4410824466E-01	3.0403009171E-01
134	1.5805075826E-01	3.5640783408E-01	5.2667402848E-01
135	1.6877853480E-01	1.8716897893E-01	5.2041032434E-01
136	3.2986998766E-01	1.7349008539E-01	5.3571406758E-01

137	3.2196656960E-01	3.4774897588E-01	5.2691975944E-01
138	1.6241740361E-01	3.1114277277E-01	6.4321023544E-01
139	1.7351139147E-01	1.9251167902E-01	6.8578253762E-01
140	3.5033875261E-01	1.6496039409E-01	6.8906200411E-01
141	3.2397395872E-01	3.0700405926E-01	6.4879962237E-01
142	5.0425714015E-01	3.3498460286E-01	5.0446401895E-01
143	5.2240052366E-01	1.7979297462E-01	5.2044836141E-01
144	7.0030733512E-01	1.9580382933E-01	5.3858694926E-01
145	6.8396487320E-01	3.1011976595E-01	5.1460594801E-01
146	5.2994296233E-01	1.6883896656E-01	7.3688375136E-01
147	5.1980765072E-01	3.0646663921E-01	6.8667479623E-01
148	6.8799537735E-01	1.7454784236E-01	6.9036588308E-01
149	6.7523611516E-01	3.1540033619E-01	6.7254242158E-01
150	2.1179449895E-01	7.3860756262E-01	4.5245943629E-01
151	1.8408689572E-01	5.4856972022E-01	5.0351016210E-01
152	3.3596269745E-01	5.2398796565E-01	5.1460194374E-01
153	3.0335689876E-01	6.8699495724E-01	4.5953541950E-01
154	1.6919881073E-01	5.2460063307E-01	7.0350810960E-01
155	3.1376435783E-01	4.6655129616E-01	6.8057361527E-01
156	1.9308223235E-01	7.0750865146E-01	6.4933162728E-01
157	3.0553270915E-01	6.4347764308E-01	6.4539869671E-01
158	5.0758984241E-01	5.0842709506E-01	5.0696244337E-01
159	6.6650276585E-01	4.7308449316E-01	5.0201940289E-01
160	4.8336966315E-01	6.9310108629E-01	4.6735717858E-01
161	6.7992359632E-01	6.8093720283E-01	4.8673222325E-01
162	5.0454810024E-01	4.8393119011E-01	6.8633254198E-01
163	6.6623251246E-01	4.6995186888E-01	6.5978563794E-01
164	4.8971087892E-01	6.7631351260E-01	6.8258141630E-01
165	6.5944555169E-01	6.5594883256E-01	6.6073271448E-01
166	8.5599488679E-01	1.4767676419E-01	1.6239814720E-01
167	8.2880410546E-01	3.4961470119E-01	1.6096172628E-01
168	8.5173535190E-01	1.2694774410E-01	3.9542470344E-01
169	8.0812489692E-01	3.3097322119E-01	3.8022978002E-01
170	1.1955689761E-01	8.8598321319E-01	1.1958000030E-01
171	3.1598828279E-01	8.5671738421E-01	1.4193223673E-01
172	1.4472373264E-01	8.5441779326E-01	3.1424414033E-01
173	3.4703043366E-01	8.0582756962E-01	3.3084965793E-01
174	6.1069861693E-01	8.6099911013E-01	1.2945980803E-01
175	8.3287888997E-01	6.4737077888E-01	1.7004225231E-01
176	8.3098598391E-01	8.5575213092E-01	1.4077550427E-01
177	6.4785177156E-01	8.2444074974E-01	3.1398906872E-01
178	8.2137660971E-01	5.8952136996E-01	3.9458027523E-01
179	8.1968375089E-01	8.0848340986E-01	3.2911221827E-01
180	1.5752757910E-01	3.5206562281E-01	8.1025859594E-01
181	1.9613499225E-01	1.4959424646E-01	8.5327433726E-01
182	3.9842315411E-01	1.4962991263E-01	8.7626194710E-01
183	3.8595942713E-01	3.4214562390E-01	8.1106658293E-01
184	8.2223541527E-01	1.5769951791E-01	6.6568668438E-01
185	8.3999150000E-01	3.2999660000E-01	6.6999320000E-01
186	6.6877608394E-01	3.1891451667E-01	8.2603146149E-01
187	6.8532675333E-01	1.5012186264E-01	8.5743240354E-01
188	8.3999150000E-01	1.5999830000E-01	8.3999150000E-01
189	8.2851625673E-01	2.7331389551E-01	8.2621256029E-01
190	8.1136538229E-01	4.5594530552E-01	6.0444994540E-01
191	6.4341488295E-01	4.3728414634E-01	7.9872708334E-01
192	8.2377699783E-01	4.1385542237E-01	8.1537444817E-01
193	8.0542540876E-01	6.2985100671E-01	6.5575299432E-01
194	6.8632264324E-01	6.2540736740E-01	8.0172718291E-01
195	8.2518512180E-01	5.8177064991E-01	8.1545899302E-01
196	1.3703083908E-01	8.4852543662E-01	6.0337815139E-01
197	3.5023445052E-01	8.1720865829E-01	6.1569206489E-01
198	1.4831694140E-01	8.4370639741E-01	8.2897843875E-01
199	1.5999830000E-01	6.6999320000E-01	8.3999150000E-01
200	3.3220416026E-01	6.0410682189E-01	8.0537811225E-01
201	3.4515508874E-01	7.9269481262E-01	8.0799987813E-01
202	6.1346577845E-01	8.1778618515E-01	6.4613419642E-01
203	8.1106370380E-01	8.2843872055E-01	6.3203026980E-01
204	6.3528805897E-01	8.2263961980E-01	8.1902596006E-01

```
  205    8.3999150000E-01    8.3999150000E-01    8.3999150000E-01
  206    9.0681114815E-01    4.7641450980E-01    4.3751606396E-01
end_nodes

cells
    0    4   16   20   52   17    0
    1    4   19   13   52   20    0
    2    4   20   19   16   52    0
    3    4   52   16   63   19    0
    4    4    0   19   62   14    0
    5    4   15   19   62   16    0
    6    4   19   14   15   62    0
    7    4   16   19   53   63    0
    8    4   16   53   19   62    0
    9    4   57   18   11   15    0
   10    4   17   15   61   16    0
   11    4   15   18   17   57    0
   12    4   17   16   10   52    0
   13    4   17   10   16   61    0
   14    4   16   15    8   62    0
   15    4   16    8   15   61    0
   16    4   13   63   60   19    0
   17    4   19   63   52   13    0
   18    4   60   63   12   19    0
   19    4   19   63   12   53    0
   20    4   10   16   63   52    0
   21    4   63   16   61   53    0
   22    4   63    9   53   61    0
   23    4   59   62    0   19    0
   24    4    0   15   62   54    0
   25    4    0   15   14   62    0
   26    4   12   62   59   19    0
   27    4   53   12   19   62    0
   28    4   15   62   54    8    0
   29    4   16   62   61   53    0
   30    4    9   62   53   61    0
   31    4   15   11   61   58    0

   32    4   15   61   11   57    0
   33    4   17   10   61   57    0
   34    4   63   61   16   10    0
   35    4   57   17   15   61    0
   36    4   61    8   15   58    0
   37    4   16   61   62    8    0
   38    4   10   63   61   56    0
   39    4   61   56   63    9    0
   40    4   55   62   61    8    0
   41    4   61    9   62   55    0
   42    4  117  112  116  115    1
   43    4  117  111  113  112    1
   44    4  115  112  111  117    1
   45    4  117  111  115  114    1
   46    4  111  114  117  113    1
   47    4  114  113  111  110    1
   48    4  125  120  124  123    1
   49    4  125  119  121  120    1
   50    4  123  120  119  125    1
   51    4  125  119  123  122    1
   52    4  119  122  125  121    1
   53    4  122  121  119  118    1
   54    4  129  113  117  114    1
   55    4  129  110  127  113    1
   56    4  114  113  110  129    1
   57    4  129  110  114  128    1
   58    4  110  128  129  127    1
   59    4  128  127  110  126    1
   60    4  131  126  128  124    1
   61    4  131  120  130  126    1
```

62	4	124	126	120	131	1
63	4	131	120	124	125	1
64	4	120	125	131	130	1
65	4	125	130	120	121	1
66	4	133	127	129	128	1
67	4	133	126	132	127	1
68	4	128	127	126	133	1
69	4	133	126	128	131	1
70	4	126	131	133	132	1
71	4	131	132	126	130	1
72	4	141	136	140	139	1
73	4	141	135	137	136	1
74	4	139	136	135	141	1
75	4	141	135	139	138	1
76	4	135	138	141	137	1
77	4	138	137	135	134	1
78	4	145	116	144	143	1
79	4	145	115	117	116	1
80	4	143	116	115	145	1
81	4	145	115	143	142	1
82	4	115	142	145	117	1
83	4	142	117	115	114	1
84	4	147	143	146	140	1
85	4	147	136	142	143	1
86	4	140	143	136	147	1
87	4	147	136	140	141	1
88	4	136	141	147	142	1
89	4	141	142	136	137	1
90	4	149	144	148	146	1
91	4	149	143	145	144	1
92	4	146	144	143	149	1
93	4	149	143	146	147	1
94	4	143	147	149	145	1
95	4	147	145	143	142	1
96	4	153	124	152	151	1
97	4	153	123	125	124	1
98	4	151	124	123	153	1
99	4	153	123	151	150	1
100	4	123	150	153	125	1
101	4	150	125	123	122	1
102	4	155	137	141	138	1
103	4	155	134	152	137	1
104	4	138	137	134	155	1
105	4	155	134	138	154	1
106	4	134	154	155	152	1
107	4	154	152	134	151	1
108	4	157	152	155	154	1
109	4	157	151	153	152	1
110	4	154	152	151	157	1
111	4	157	151	154	156	1
112	4	151	156	157	153	1
113	4	156	153	151	150	1
114	4	159	117	145	142	1
115	4	159	114	129	117	1
116	4	142	117	114	159	1
117	4	159	114	142	158	1
118	4	114	158	159	129	1
119	4	158	129	114	128	1
120	4	160	128	158	152	1
121	4	160	124	131	128	1
122	4	152	128	124	160	1
123	4	160	124	152	153	1
124	4	124	153	160	131	1
125	4	153	131	124	125	1
126	4	161	129	159	158	1
127	4	161	128	133	129	1
128	4	158	129	128	161	1
129	4	161	128	158	160	1

130	4	128	160	161	133	1
131	4	160	133	128	131	1
132	4	162	142	147	141	1
133	4	162	137	158	142	1
134	4	141	142	137	162	1
135	4	162	137	141	155	1
136	4	137	155	162	158	1
137	4	155	158	137	152	1
138	4	163	145	149	147	1
139	4	163	142	159	145	1
140	4	147	145	142	163	1
141	4	163	142	147	162	1
142	4	142	162	163	159	1
143	4	162	159	142	158	1
144	4	164	158	162	155	1
145	4	164	152	160	158	1
146	4	155	158	152	164	1
147	4	164	152	155	157	1
148	4	152	157	164	160	1
149	4	157	160	152	153	1
150	4	165	159	163	162	1
151	4	165	158	161	159	1
152	4	162	159	158	165	1
153	4	165	158	162	164	1
154	4	158	164	165	161	1
155	4	164	161	158	160	1
156	4	169	166	168	116	1
157	4	169	112	167	166	1
158	4	116	166	112	169	1
159	4	169	112	116	117	1
160	4	112	117	169	167	1
161	4	117	167	112	113	1
162	4	173	121	125	122	1
163	4	173	118	171	121	1
164	4	122	121	118	173	1
165	4	173	118	122	172	1
166	4	118	172	173	171	1
167	4	172	171	118	170	1
168	4	179	175	178	133	1
169	4	179	132	176	175	1
170	4	133	175	132	179	1
171	4	179	132	133	177	1
172	4	132	177	179	176	1
173	4	177	176	132	174	1
174	4	183	140	182	181	1
175	4	183	139	141	140	1
176	4	181	140	139	183	1
177	4	183	139	181	180	1
178	4	139	180	183	141	1
179	4	180	141	139	138	1
180	4	189	184	188	187	1
181	4	189	148	185	184	1
182	4	187	184	148	189	1
183	4	189	148	187	186	1
184	4	148	186	189	185	1
185	4	186	185	148	149	1
186	4	192	185	189	186	1
187	4	192	149	190	185	1
188	4	186	185	149	192	1
189	4	192	149	186	191	1
190	4	149	191	192	190	1
191	4	191	190	149	163	1
192	4	195	190	192	191	1
193	4	195	163	193	190	1
194	4	191	190	163	195	1
195	4	195	163	191	194	1
196	4	163	194	195	193	1
197	4	194	193	163	165	1

198	4	201	157	200	199	1
199	4	201	156	197	157	1
200	4	199	157	156	201	1
201	4	201	156	199	198	1
202	4	156	198	201	197	1
203	4	198	197	156	196	1
204	4	205	193	195	194	1
205	4	205	165	203	193	1
206	4	194	193	165	205	1
207	4	205	165	194	204	1
208	4	165	204	205	203	1
209	4	204	203	165	202	1
210	4	52	63	111	13	1
211	4	82	13	111	60	1
212	4	13	60	63	111	1
213	4	13	50	82	71	1
214	4	50	71	112	82	1
215	4	77	50	71	112	1
216	4	60	63	110	12	1
217	4	60	82	110	111	1
218	4	111	60	63	110	1
219	4	82	12	60	110	1
220	4	113	82	111	110	1
221	4	113	112	111	82	1
222	4	82	83	113	112	1
223	4	10	63	115	52	1
224	4	52	115	10	76	1
225	4	63	115	52	111	1
226	4	63	56	10	115	1
227	4	111	52	71	115	1
228	4	13	111	52	71	1
229	4	52	71	115	76	1
230	4	115	71	112	116	1
231	4	115	111	112	71	1
232	4	114	63	111	115	1
233	4	63	110	114	111	1
234	4	115	114	63	56	1
235	4	63	9	56	114	1
236	4	63	114	53	9	1
237	4	59	62	119	0	1
238	4	81	59	119	0	1
239	4	51	0	81	119	1
240	4	12	62	120	59	1
241	4	81	12	120	59	1
242	4	62	120	59	119	1
243	4	59	120	81	119	1
244	4	119	51	0	69	1
245	4	81	119	118	51	1
246	4	69	51	118	119	1
247	4	81	80	51	118	1
248	4	121	81	119	118	1
249	4	121	119	81	120	1
250	4	118	80	121	81	1
251	4	62	8	123	54	1
252	4	69	54	123	8	1
253	4	62	8	55	123	1
254	4	69	54	119	123	1
255	4	69	54	0	119	1
256	4	123	62	120	124	1
257	4	62	119	123	120	1
258	4	124	123	62	55	1
259	4	120	53	124	62	1
260	4	124	9	62	53	1
261	4	62	124	55	9	1
262	4	122	69	119	123	1
263	4	122	119	69	118	1
264	4	123	67	122	69	1
265	4	62	120	53	12	1

266	4	12	53	126	120	1
267	4	120	12	79	126	1
268	4	12	84	113	82	1
269	4	127	84	110	126	1
270	4	12	84	110	113	1
271	4	127	113	110	84	1
272	4	12	84	126	110	1
273	4	84	79	12	126	1
274	4	81	120	12	79	1
275	4	130	120	79	126	1
276	4	81	79	121	120	1
277	4	79	80	81	121	1
278	4	132	79	126	130	1
279	4	126	127	84	79	1
280	4	132	127	126	79	1
281	4	79	84	132	127	1
282	4	128	53	110	114	1
283	4	124	53	126	128	1
284	4	114	9	128	53	1
285	4	124	120	126	53	1
286	4	128	126	110	53	1
287	4	53	124	9	128	1
288	4	58	61	135	11	1
289	4	11	135	58	70	1
290	4	11	57	135	72	1
291	4	11	57	61	135	1
292	4	72	11	70	135	1
293	4	22	72	11	70	1
294	4	57	61	136	10	1
295	4	10	136	57	74	1
296	4	57	72	136	135	1
297	4	135	61	136	57	1
298	4	74	57	72	136	1
299	4	8	58	134	65	1
300	4	8	58	61	134	1
301	4	58	134	135	61	1
302	4	137	61	135	134	1
303	4	137	135	61	136	1
304	4	134	55	137	61	1
305	4	137	9	61	55	1
306	4	137	56	61	9	1
307	4	61	137	136	56	1
308	4	135	72	139	70	1
309	4	70	22	72	139	1
310	4	139	72	73	22	1
311	4	74	73	140	72	1
312	4	139	72	136	140	1
313	4	135	139	72	136	1
314	4	72	74	136	140	1
315	4	70	65	134	58	1
316	4	138	70	135	139	1
317	4	70	65	138	134	1
318	4	138	134	135	70	1
319	4	70	135	58	134	1
320	4	61	136	10	56	1
321	4	115	10	143	56	1
322	4	10	74	143	136	1
323	4	10	143	56	136	1
324	4	76	10	74	143	1
325	4	143	76	116	144	1
326	4	143	76	10	115	1
327	4	74	143	144	76	1
328	4	116	168	144	76	1
329	4	76	74	148	144	1
330	4	142	115	56	114	1
331	4	142	136	56	143	1
332	4	114	56	142	9	1

333	4	9	142	137	56	1
334	4	142	56	115	143	1
335	4	142	136	137	56	1
336	4	140	74	143	146	1
337	4	140	143	74	136	1
338	4	146	143	144	74	1
339	4	76	75	148	74	1
340	4	146	74	144	148	1
341	4	144	76	184	148	1
342	4	148	75	146	74	1
343	4	61	134	55	8	1
344	4	8	123	151	55	1
345	4	55	151	8	134	1
346	4	65	134	8	151	1
347	4	151	124	55	123	1
348	4	152	134	55	137	1
349	4	9	152	124	55	1
350	4	137	55	152	9	1
351	4	152	55	134	151	1
352	4	151	124	152	55	1
353	4	8	67	123	69	1
354	4	150	123	67	122	1
355	4	122	68	69	67	1
356	4	8	67	151	123	1
357	4	8	67	65	151	1
358	4	154	65	134	138	1
359	4	154	151	134	65	1
360	4	138	180	154	65	1
361	4	65	66	154	67	1
362	4	156	67	151	154	1
363	4	151	154	67	65	1
364	4	66	67	156	154	1
365	4	158	9	114	142	1
366	4	152	9	128	158	1
367	4	158	9	137	152	1
368	4	158	114	9	128	1
369	4	128	124	9	152	1
370	4	112	50	166	77	1
371	4	112	50	83	166	1
372	4	82	112	50	83	1
373	4	77	7	166	25	1
374	4	25	86	7	166	1
375	4	84	113	82	83	1
376	4	167	112	83	166	1
377	4	167	112	113	83	1
378	4	166	167	47	83	1
379	4	84	47	167	83	1
380	4	86	166	167	47	1
381	4	89	47	86	167	1
382	4	76	77	168	24	1
383	4	116	166	77	112	1
384	4	76	116	168	77	1
385	4	116	166	168	77	1
386	4	169	168	166	86	1
387	4	86	89	169	91	1
388	4	166	169	86	167	1
389	4	167	169	86	89	1
390	4	86	168	91	169	1
391	4	84	167	47	48	1
392	4	48	84	175	78	1
393	4	48	85	175	89	1
394	4	129	113	167	117	1
395	4	133	127	175	129	1
396	4	117	167	129	169	1
397	4	133	175	127	132	1
398	4	84	167	127	113	1
399	4	175	127	132	84	1
400	4	84	113	83	167	1

401	4	85	89	178	175	1
402	4	127	129	89	175	1
403	4	167	169	89	129	1
404	4	129	113	127	167	1
405	4	129	89	175	178	1
406	4	167	127	129	89	1
407	4	169	129	178	89	1
408	4	89	167	127	175	1
409	4	23	75	184	76	1
410	4	91	88	184	23	1
411	4	23	24	168	91	1
412	4	145	116	169	144	1
413	4	149	144	185	148	1
414	4	145	169	116	117	1
415	4	116	169	144	168	1
416	4	144	76	168	184	1
417	4	76	23	24	168	1
418	4	24	86	168	91	1
419	4	76	148	75	184	1
420	4	91	168	23	184	1
421	4	149	144	145	185	1
422	4	144	91	168	169	1
423	4	76	168	184	23	1
424	4	190	185	149	145	1
425	4	92	91	190	90	1
426	4	159	117	169	145	1
427	4	161	129	178	159	1
428	4	163	145	190	149	1
429	4	165	159	193	163	1
430	4	159	193	163	190	1
431	4	159	169	117	129	1
432	4	129	159	169	178	1
433	4	178	159	169	190	1
434	4	145	185	169	190	1
435	4	90	92	193	190	1
436	4	161	129	133	178	1
437	4	165	159	161	193	1
438	4	178	190	193	159	1
439	4	190	145	159	169	1
440	4	90	178	190	193	1
441	4	159	193	178	161	1
442	4	193	161	90	178	1
443	4	69	68	118	51	1
444	4	118	51	170	80	1
445	4	68	51	170	118	1
446	4	5	68	51	170	1
447	4	171	80	118	170	1
448	4	171	121	118	80	1
449	4	79	80	121	171	1
450	4	31	5	170	68	1
451	4	46	80	170	5	1
452	4	5	170	46	93	1
453	4	51	170	80	5	1
454	4	93	31	5	170	1
455	4	45	80	171	46	1
456	4	170	93	171	80	1
457	4	46	80	93	170	1
458	4	80	79	45	171	1
459	4	80	171	46	93	1
460	4	172	68	118	122	1
461	4	69	122	118	68	1
462	4	172	118	68	170	1
463	4	122	67	172	68	1
464	4	30	31	172	68	1
465	4	68	31	93	170	1
466	4	170	93	68	172	1
467	4	93	30	31	172	1
468	4	31	172	68	93	1

469	4	98	97	173	93	1
470	4	172	171	93	173	1
471	4	172	171	170	93	1
472	4	171	173	97	93	1
473	4	44	79	174	45	1
474	4	45	93	171	97	1
475	4	45	174	44	97	1
476	4	125	130	173	131	1
477	4	131	132	177	133	1
478	4	173	121	130	125	1
479	4	130	174	132	79	1
480	4	171	130	121	79	1
481	4	171	45	174	79	1
482	4	171	173	130	97	1
483	4	45	174	97	171	1
484	4	130	131	97	173	1
485	4	131	132	130	177	1
486	4	132	130	177	174	1
487	4	173	121	171	130	1
488	4	171	174	97	130	1
489	4	131	173	177	97	1
490	4	130	174	97	177	1
491	4	131	173	160	177	1
492	4	130	97	131	177	1
493	4	29	30	196	67	1
494	4	172	68	67	30	1
495	4	98	93	172	30	1
496	4	98	30	196	29	1
497	4	150	125	173	153	1
498	4	156	153	197	157	1
499	4	173	122	150	172	1
500	4	197	150	156	196	1
501	4	122	150	172	67	1
502	4	66	67	196	156	1
503	4	98	172	93	173	1
504	4	67	172	30	196	1
505	4	98	196	30	172	1
506	4	150	125	122	173	1
507	4	157	160	153	197	1
508	4	156	197	153	150	1
509	4	67	29	66	196	1
510	4	67	196	150	172	1
511	4	150	98	197	153	1
512	4	67	196	156	150	1
513	4	153	173	150	98	1
514	4	197	153	98	173	1
515	4	98	173	97	99	1
516	4	100	99	197	98	1
517	4	153	131	173	160	1
518	4	160	133	177	161	1
519	4	157	160	197	164	1
520	4	164	202	161	160	1
521	4	153	131	125	173	1
522	4	173	177	99	160	1
523	4	153	173	197	160	1
524	4	99	197	98	173	1
525	4	100	95	98	197	1
526	4	161	160	99	177	1
527	4	164	157	201	197	1
528	4	161	99	160	202	1
529	4	173	177	97	99	1
530	4	160	197	202	99	1
531	4	177	99	94	97	1
532	4	160	99	173	197	1
533	4	44	174	79	78	1
534	4	176	78	132	174	1
535	4	78	79	132	174	1
536	4	78	79	84	132	1

537	4	78	175	48	49	1
538	4	49	175	48	85	1
539	4	176	78	6	49	1
540	4	97	94	174	44	1
541	4	44	78	176	174	1
542	4	94	44	176	174	1
543	4	6	94	44	176	1
544	4	176	6	78	44	1
545	4	49	6	176	85	1
546	4	176	94	28	6	1
547	4	6	176	85	28	1
548	4	89	178	90	85	1
549	4	179	85	175	176	1
550	4	179	175	85	178	1
551	4	177	176	94	179	1
552	4	177	174	97	94	1
553	4	177	174	94	176	1
554	4	179	177	99	94	1
555	4	85	28	179	27	1
556	4	28	179	27	94	1
557	4	90	27	85	179	1
558	4	90	27	203	26	1
559	4	94	179	27	99	1
560	4	27	203	26	99	1
561	4	179	133	161	177	1
562	4	203	161	165	202	1
563	4	178	161	179	133	1
564	4	178	179	161	90	1
565	4	177	161	179	99	1
566	4	85	90	179	178	1
567	4	179	90	27	203	1
568	4	99	179	27	203	1
569	4	165	164	202	161	1
570	4	161	99	203	179	1
571	4	202	161	99	203	1
572	4	22	64	139	70	1
573	4	22	139	181	73	1
574	4	22	64	181	139	1
575	4	4	73	22	181	1
576	4	181	73	140	182	1
577	4	140	139	72	73	1
578	4	73	140	139	181	1
579	4	140	146	182	73	1
580	4	64	42	180	65	1
581	4	180	139	64	138	1
582	4	65	64	138	180	1
583	4	64	4	101	43	1
584	4	73	4	38	101	1
585	4	22	181	64	4	1
586	4	181	64	4	101	1
587	4	4	181	101	73	1
588	4	102	101	181	38	1
589	4	101	181	38	73	1
590	4	38	102	182	181	1
591	4	107	38	102	182	1
592	4	64	180	42	43	1
593	4	43	180	42	102	1
594	4	183	181	102	180	1
595	4	183	102	181	182	1
596	4	39	74	146	75	1
597	4	73	39	182	38	1
598	4	39	104	187	107	1
599	4	39	182	38	107	1
600	4	147	140	183	141	1
601	4	149	146	186	147	1
602	4	183	140	146	182	1
603	4	186	148	146	149	1

604	4	148	187	146	75	1
605	4	73	182	39	146	1
606	4	182	183	102	107	1
607	4	39	74	73	146	1
608	4	104	107	186	187	1
609	4	107	187	39	182	1
610	4	146	147	107	186	1
611	4	147	183	140	146	1
612	4	75	146	39	187	1
613	4	182	107	146	183	1
614	4	108	104	107	186	1
615	4	182	146	107	187	1
616	4	146	147	183	107	1
617	4	186	146	187	107	1
618	4	182	187	39	146	1
619	4	42	66	154	65	1
620	4	102	180	42	106	1
621	4	41	42	199	106	1
622	4	155	138	180	154	1
623	4	157	154	199	156	1
624	4	180	141	155	183	1
625	4	157	199	154	200	1
626	4	65	154	42	180	1
627	4	154	66	199	156	1
628	4	42	66	199	154	1
629	4	106	41	103	199	1
630	4	180	141	138	155	1
631	4	66	41	42	199	1
632	4	200	199	154	106	1
633	4	107	108	191	109	1
634	4	162	141	183	155	1
635	4	163	147	191	162	1
636	4	164	155	200	157	1
637	4	165	162	194	164	1
638	4	191	147	149	186	1
639	4	194	162	163	191	1
640	4	162	141	147	183	1
641	4	107	186	191	108	1
642	4	108	109	194	191	1
643	4	155	162	200	183	1
644	4	147	162	183	191	1
645	4	157	154	155	200	1
646	4	162	164	109	194	1
647	4	191	163	149	147	1
648	4	109	105	108	194	1
649	4	164	155	162	200	1
650	4	186	191	147	183	1
651	4	164	194	204	109	1
652	4	162	200	109	164	1
653	4	162	200	183	109	1
654	4	109	162	191	183	1
655	4	187	75	184	188	1
656	4	187	184	75	148	1
657	4	184	23	188	75	1
658	4	188	75	23	3	1
659	4	88	23	188	184	1
660	4	3	88	23	188	1
661	4	88	91	185	92	1
662	4	189	184	88	188	1
663	4	189	184	185	88	1
664	4	88	189	92	185	1
665	4	75	40	187	39	1
666	4	40	187	39	104	1
667	4	75	40	104	187	1
668	4	188	187	75	104	1
669	4	189	187	104	186	1
670	4	186	104	189	108	1
671	4	189	187	188	104	1

672	4	35	88	189	92	1
673	4	188	35	88	189	1
674	4	189	35	104	188	1
675	4	192	185	92	189	1
676	4	192	185	190	92	1
677	4	92	192	195	190	1
678	4	90	193	92	87	1
679	4	190	195	92	193	1
680	4	92	192	36	195	1
681	4	87	92	195	193	1
682	4	92	36	87	195	1
683	4	192	186	108	191	1
684	4	192	108	186	189	1
685	4	195	191	192	108	1
686	4	92	36	192	35	1
687	4	36	192	35	108	1
688	4	35	192	92	189	1
689	4	108	35	189	192	1
690	4	108	195	191	194	1
691	4	194	108	195	105	1
692	4	36	192	108	195	1
693	4	36	105	195	108	1
694	4	198	66	156	199	1
695	4	198	156	66	196	1
696	4	199	41	198	66	1
697	4	1	29	198	66	1
698	4	29	95	196	98	1
699	4	66	198	196	29	1
700	4	29	95	198	196	1
701	4	95	1	29	198	1
702	4	198	197	95	201	1
703	4	197	98	196	95	1
704	4	95	196	197	198	1
705	4	198	66	41	1	1
706	4	199	198	41	103	1
707	4	1	103	41	198	1
708	4	103	106	200	109	1
709	4	201	199	103	198	1
710	4	199	106	200	103	1
711	4	201	199	200	103	1
712	4	95	1	198	34	1
713	4	198	103	34	1	1
714	4	100	95	201	33	1
715	4	201	33	103	109	1
716	4	32	33	204	109	1
717	4	204	164	165	194	1
718	4	100	197	201	95	1
719	4	197	164	100	201	1
720	4	204	202	165	164	1
721	4	164	201	157	200	1
722	4	109	105	194	204	1
723	4	103	201	109	200	1
724	4	33	204	109	201	1
725	4	109	32	105	204	1
726	4	202	100	164	204	1
727	4	200	109	164	201	1
728	4	202	197	164	100	1
729	4	201	204	109	164	1
730	4	26	87	203	90	1
731	4	205	87	193	203	1
732	4	87	203	90	193	1
733	4	205	195	193	87	1
734	4	99	96	203	26	1
735	4	96	99	202	100	1
736	4	204	203	96	205	1
737	4	100	96	204	202	1
738	4	205	204	32	96	1
739	4	96	202	203	204	1

740	4	96	202	99	203	1
741	4	26	87	205	203	1
742	4	96	26	205	203	1
743	4	205	194	105	204	1
744	4	205	105	194	195	1
745	4	205	32	204	105	1
746	4	87	37	105	36	1
747	4	195	87	105	36	1
748	4	195	105	87	205	1
749	4	204	32	96	100	1
750	4	71	82	13	111	1
751	4	111	71	82	112	1
752	4	63	12	53	110	1
753	4	113	110	12	82	1
754	4	116	71	76	115	1
755	4	110	53	63	114	1
756	4	62	54	119	0	1
757	4	62	119	54	123	1
758	4	12	53	110	126	1
759	4	130	79	120	121	1
760	4	143	116	76	115	1
761	4	73	140	146	74	1
762	4	150	67	123	151	1
763	4	156	151	67	150	1
764	4	158	137	9	142	1
765	4	48	167	47	89	1
766	4	132	84	78	175	1
767	4	127	84	175	167	1
768	4	175	129	178	133	1
769	4	144	185	148	184	1
770	4	88	185	91	184	1
771	4	91	184	144	168	1
772	4	184	185	91	144	1
773	4	163	190	145	159	1
774	4	185	190	91	169	1
775	4	92	190	91	185	1
776	4	46	171	45	93	1
777	4	160	133	131	177	1
778	4	79	174	171	130	1
779	4	98	172	173	150	1
780	4	150	98	172	196	1
781	4	150	197	98	196	1
782	4	99	202	100	197	1
783	4	160	197	164	202	1
784	4	176	132	78	175	1
785	4	203	165	161	193	1
786	4	90	161	203	179	1
787	4	193	90	161	203	1
788	4	181	182	38	73	1
789	4	180	64	139	181	1
790	4	180	183	106	102	1
791	4	186	146	148	187	1
792	4	147	186	183	107	1
793	4	107	191	186	183	1
794	4	107	106	183	102	1
795	4	154	155	106	180	1
796	4	154	106	155	200	1
797	4	183	155	180	106	1
798	4	200	183	106	155	1
799	4	107	183	106	109	1
800	4	194	163	162	165	1
801	4	109	191	162	194	1
802	4	109	183	106	200	1
803	4	107	183	109	191	1
804	4	108	35	104	189	1
805	4	100	33	204	32	1
806	4	201	204	164	100	1

```
    807    4   33  201  100  204    1
    808    4  169  206  178  190    1
    809    4  169  178  206   89    1
    810    4   90  206   91  190    1
    811    4   90   91  206   89    1
    812    4   90  206  178   89    1
    813    4   90  178  206  190    1
    814    4  169  206   91   89    1
    815    4  169   91  206  190    1
    816    4  112  116   71   77    1
    817    4  116   76   71   77    1
    818    4   50    7   83  166    1
    819    4   50   77    7  166    1
    820    4  166   86    7   47    1
    821    4   83  166    7   47    1
    822    4   77   24  166  168    1
    823    4  166  168   24   86    1
    824    4   24   25   77  166    1
    825    4   25  166   24   86    1
    826    4   84  167   48  175    1
    827    4   48  167   89  175    1
    828    4  169  144   91  185    1
    829    4  144  185  169  145    1
    830    4  175   78  176   49    1
    831    4   85  175  176   49    1
    832    4  176   85   28  179    1
    833    4  176   28   94  179    1
    834    4  181  101  180   64    1
    835    4  181  180  101  102    1
    836    4   43  101   64  180    1
    837    4   43  102  101  180    1
    838    4  180  106  154   42    1
    839    4   42  106  154  199    1
    840    4  188    3   88   35    1
    841    4    3  188  104   35    1
    842    4  201   95   34   33    1
    843    4  103  201   34   33    1
    844    4  201   95  198   34    1
    845    4  198  103  201   34    1
    846    4   96  205   26    2    1
    847    4   26  205   87    2    1
    848    4  105   87  205    2    1
    849    4   87   37    2  105    1
    850    4   96   32  205    2    1
    851    4  205   32  105    2    1
    852    4  139   64  138   70    1
    853    4   64   70   65  138    1
    854    4   75   40    3  104    1
    855    4   75  104    3  188    1
    856    4   20   14   15   19    0
    857    4   14   21   20   15    0
    858    4   20   17   15   18    0
    859    4   21   15   18   20    0
    860    4   16   15   20   17    0
    861    4   19   15   20   16    0
end_cells
```

```
  ****** Frost - FiRst Order Sn Transport ********

$Id: README,v 1.16 2004/03/10 17:17:26 mcghee Exp $
```

```
This a multi-group 3D arbitrary tetrahedral mesh discrete ordinates
demonstration code written in Python. The purpose of this code is to
demonstrate the general features of the solution algorithm. It is not
optimized for computational efficiency and is not intended to be used
for large problems or in a production environment.

Written by:

 John McGhee, Radion Technologies, Los Alamos NM, 19 Feb 2004.

This program is written in Python. Python is a widely available,
powerful, easy to use, and easy to learn scripting language.
(And it's free!) References:
 -- "Learning Python", Mark Lutz and David Ascher, O'Reilly, 1999.
 -- "Python Essential Reference", David Beazley, New Riders, 2000.
 --  Python Home Page -- http://http://www.python.org/
 --  The Vaults of Parnassus -- http://www.vex.net/parnassus/
 --  Numerical Python -- http://www.pfdubois.com/numpy/
 --  Scipy -- http://www.scipy.org/
 --  Starship Python - http://starship.python.net

A discussion of the design philosophy of the code and various other
background information is provided in the file background_info.asc.
Extensive comments are provided in the body of the source code that
explain details of the algorithm.

A small sample problem is included for demonstration purposes. All
output is written to the screen with the exception of a GMV
visualization link file. Sample screen output is included in the file
"frost.out_std". A compressed GMV output file is included as
"gmv.out_std.gz". GMV is a publicly available visualization utility
that can be obtained from Los Alamos National Laboratory, Los Alamos NM.
See http://www-xdiv.lanl.gov/XCM/gmv/

How to run the code:
Simply type ./frost.py at the command prompt to execute the included
sample problem. Runs in about two minutes, sending output to the screen as
the calculation progresses. Other problems can be run by replacing the
data files:
  matprop.inp,

  aquad.inp, and,
  mesh.inp
with the data files of choice and redefining the fixed source as desired
in the file "fsrc.py".  Formats for the data files are briefly defined
in the included demo files. Adding the command line option "fsds" will
cause the first scattered distributed source solution demonstration mode
to be invoked.

Limitations and things Frost will not do:

 -- tets only, no other spatial element types
 -- standard scattering only, no Galerkin scattering
 -- P1 scattering only, no higher order scattering terms
 -- no meshes which are not directed acyclic graphs (DAG)
 -- only vacuum boundary conditions
 -- no dsa of scattering source
 -- fsds option assumes a single isotropic point source. Only first
    scattered source problems are supported, no second or subsequent
    collided sources.
 -- no upscatter in energy, down-scatter only.
 -- only a one-to-one correspondence between mesh regions and materials.
    is supported, no material mixing.

Things that are included in Frost:

-- Geometry Class - geom.py - Reads in mesh off disk, computes tet
```

```
volumes, face areas, face normals, connectivity, pointers to adjacent
cells. Provides accessors methods to get to it all.

-- Angular Quadrature Class - aquad.py - Reads in quadrature off disk.
Accessor methods to get at data. Computes discrete to moment and moment
to discrete matrices. Only does standard scattering, no Galerkin.

-- Fixed Source Class - fsrc.py - Defines volumetric fixed sources.
Provides accessor methods to get at the data.

-- BteEquation Class - bte.py - Forms left and right hand
side of LD Boltzmann transport equation for neutral and charged
particles. Solves  the system with LU decomp.

 -- Frost - frost.py - Main driver. Calls all the methods described above,
loops through groups, angles, and mesh cells.  Computes balance table for
edit and debug. Writes visualization dump.

-- Sweep Ordering Class - sord.py - Orders the mesh cells from upwind
to downwind for any specified quadrature direction. Does not do graph
cycles.

-- Spherical Harmonics methods - sph.py - Evaluates the orthonormal
spherical harmonics surface functions for any given direction. Only
good through degree 1.

-- Material Properties Class - matp.py - Reads in multi-group material
properties from disk. Provides accessor functions to get at the data.

-- Edits Class - edit.py - Writes GMV link file for visualization. Useful
for debug and other post-processing.

-- Scattering Methods - scat.py - Does out of group scatter and self scatter
separately, iterates to convergence with good particle balance.

-- Charged Particles - This was accomplished as an addition
to the BteEquation class.

-- First Scattered Distributed Source Class - fsds.py provides an
analytic method to calculate the uncollided solution component and
computes a first scattered distributed source. Also provides a method
to add the uncollided and collided components together to form the
total solution.

-- Ray Tracing Class - trace.py provides algorithms for tracing rays
on tetrahedral meshes, related algorithms for interpolation of
fields, and methods for finding cells which contain arbitrarily
located spatial points.


                                    -- jmm, Los Alamos, 10 Mar 2004.
#------------------------------------------------------------------
#
# File: scat.py
#
#      Creates scattering source moments.
#
# Author: John McGhee, Radion Technologies
#
# Date  : 19 Feb 2004

#------------------------------------------------------------------
#
# $Id: scat.py,v 1.6 2004/03/05 23:01:13 mcghee Exp $
#
#------------------------------------------------------------------

from sys import exit
```

```
def in_scatter(icell, ig, meshdata, matprop, momdata, qdata):
    "Produces the in_scatter source for cell icell and group ig)"

    result = []
    isink = ig
    imat = meshdata.region(icell)
    for imom in range(qdata.nmom):
        l = qdata.index_lm[imom][0]
        x = [ 0., 0., 0., 0.]
        for isrc in range(matprop.ngroups):
            if (isrc == isink):
                continue
            for j in range(4):
                x[j] = x[j] + momdata[imom][isrc][icell][j]* \
                       matprop.sigsct(l, isrc, isink, imat)
        result.append(x)
    return result

# ---------------------------------------------------------------------

def self_scatter(icell, ig, meshdata, matprop, momdata, qdata):
    "Produces the self_scatter source for cell icell and group ig)"

    result = []
    isrc  = ig
    isink = ig
    imat = meshdata.region(icell)
    for imom in range(qdata.nmom):
        l = qdata.index_lm[imom][0]
        x = []
        for j in range(4):
            x.append(momdata[imom][ig][icell][j]* \
                   matprop.sigsct(l, isrc, isink, imat) )
        result.append(x)
    return result

#---------------------------------------------------------------------
#                        end of scat.py
#---------------------------------------------------------------------
#---------------------------------------------------------------------
#
# File: sord.py
#
#      Determines the ordering of the cell solution process
#      for a particular direction and mesh.
#
#      This demo assumes a tet mesh, but other arbitrary finite
#      element meshes can be ordered in a similar manner. For elements
#      with curvilinear faces, a face average normal is often used
#      to determine in/out flow status of the face.
#
#      The Sn sweep dependencies can be defined in terms of a directed graph.
#      We define the vertexes of the graph to be the mesh cells, and the
#      edges of the graph to be the outgoing faces on a cell directed to
#      the incoming faces on the adjacent cells. We get a different graph for

#      every Sn angle on every mesh.
#
#      The process of finding a sweep ordering is what is known as a
#      topological sort in graph theory. We hope that the sweep-ordering for
#      each angle can be represented as a directed-acyclic-graph (dag).
#      If strongly connected components (cycles) are present,
#      then the mesh graph is not a dag and no topological sort is
#      possible.
#
#      If cycles are discovered, a graph theory method known as a
#      "depth-first-search" can be used to convert the graph into a
```

```
#       dag. A depth first search partitions the
#       graph edges into four types: tree-edges, forward-edges, cross-edges,
#       and back-edges. Viewed from any vertex, the depth first search for
#       a dag has no back-edges. Thus the graph can be converted to a
#       dag by removing the back edges.
#
#       For purposes of simplification, the algorithm herein assumes
#       that the mesh graph is a dag.
#
#
# Author: John McGhee, Radion Technologies
#
# Date  : 19 Feb 2004

#-----------------------------------------------------------------------
#
# $Id: sord.py,v 1.6 2004/03/05 21:51:31 mcghee Exp $
#
#-----------------------------------------------------------------------

from sys import exit


class SolutionOrder:
    "Provides information on the mesh cell ordering required for solution."

    def __init__(self, geom, quad):
        "Sets up the sweep ordering"
        self.ncells = geom.ncells
        self.nang = quad.nang
        self.sweep_order = []

        # Loop over angles
        for kk in xrange(quad.nang):
            nnd = self.ncells
            omega = quad.qdpt(kk)
            self.sweep_order.append([])
            faces_needed = []
            out_face = []

            # Find the outgoing cell faces for this angle.
            for icell in xrange(self.ncells):
                i = 0
                out_face.append([])
                for iface in range(geom.nfaces):
                    xx = 0.
                    yy = geom.avect(icell,iface)
                    for j in range(geom.ndim):
                        xx = xx + omega[j]*yy[j]
                    if ( ( xx < -geom.dplimit ) and
                         ( geom.adjcell(icell,iface)[0] != geom.bdry_flag[0] ) ):
                            i = i+1
                    elif ( xx > geom.dplimit ):
                        out_face[icell].append(iface)
                faces_needed.append(i)
                if ( i == 0 ):
                    self.sweep_order[kk].append(icell)
                    nnd = nnd - 1

            # Go through the mesh in a wavefront manner, ordering the
            # cells by wave front number.
            i = 0
            this_wave = self.sweep_order[kk][:]
            if (len(this_wave) == 0):
                print "Error: unable to find any starting cells."
                print "kk= ", kk, ", nnd= ", nnd, ", swp= ", i
                exit()
            while (nnd > 0):
```

```
                    i = i+1
                    if ( i >= self.ncells):
                        print "Error: number of sweeps exceeds ncells!"
                        exit()
                    next_wave = []
                    for icell in this_wave:
                        for iface in out_face[icell]:
                            jcell = geom.adjcell(icell,iface)[0]
                            if (jcell == geom.bdry_flag[0]):
                                continue
                            faces_needed[jcell] = faces_needed[jcell]  - 1
                            if (faces_needed[jcell] == 0):
                                next_wave.append(jcell)
                                self.sweep_order[kk].append(jcell)
                                nnd = nnd - 1
                    if (len(next_wave) == 0):
                        # At this point a cycle in the mesh graph has been
                        # found. A production algorithm would call on a
                        # method to break this cycle by flagging back edges
                        # in the mesh for iteration.
                        print "Error: graph cycle found, unable to continue sweep
ordering process."
                        print "kk= ", kk, ", nnd= ", nnd, ", swp= ", i
                        exit()
                    this_wave = next_wave

                if ( len(self.sweep_order[kk]) != self.ncells ):
                    print "Error: did not order all cells."
                    exit()

                # return for next angle.

        # ---------------------------------------------------------------------

        def order(self,kk):
            "Returns the cell-wise order of solution for angle kk"
            if (kk < 0) or (kk > self.nang):
                print "Error: angle argument out of range in cell_order"
                exit()
            return self.sweep_order[kk]

    #---------------------------------------------------------------------
    #                       end of sord.py
    #---------------------------------------------------------------------
    #---------------------------------------------------------------------
    #
    # File: sph.py
    #
    #       Evaluates the orthonormal spherical harmonics surface
    #       functions. This function is defined in standard
    #       mathematical references. Normalization used herein
    #       assumes integration over the unit sphere = 1.
    #       This demo function generator only supports degree
    #       <= 1, a production version would support any
    #       requested degree and order.
    #
    #       l is the degree of the function (l>=0)
    #       m is the order of the function (-l<=m<=l)
    #       xmu is the cosine of the polar angle. The azimuthal angle
    #       phi is measured in the plus xi direction from the plus eta axis.

    #       sph = [p(l,abs(m),xmu)]*[cos(abs(m)*phi)] for m.ge.0;
    #           = [p(l,abs(m),xmu)]*[sin(abs(m)*phi)] for m.lt.0

    #       where p(l,m,xmu) is the Associated Legendre Polynomial

    #       m = 0 returns the Legendre Polynomials
    #
```

```
# Author: John McGhee, Radion Technologies
#
# Date  : 19 Feb 2004

#-----------------------------------------------------------------------
#
# $Id: sph.py,v 1.6 2004/03/05 21:51:31 mcghee Exp $
#
#-----------------------------------------------------------------------

def sphfun(l, m, xmu, eta, xi):
    "Evaluates the ortho-normal spherical harmonics surface function"

    from sys import exit
    from math import sqrt, cos, sin, acos, pi

    if ( l < 0 ):
        print "Error: l must be >= 0 in sph"
        exit()
    if ( abs(m) > l ):
        print "Error: abs(m) must be <= l"
        exit()

    if (l == 0):
        return 1.
    elif (l == 1):
        if (m == 0):
            return sqrt(3.)*xmu
        else:
            # Catch the limiting case of xmu = +1,-1
            xx = min ( (1.-xmu)*(1.+xmu), (eta*eta + xi*xi) )
            if (xx < 1.e-16):
                return 0.

            # Get the azimuthal angle, phi
            xx = eta*eta
            xx = xx/(xx + xi*xi)
            xx = max( 0., min(1., xx) )
            xx = acos( sqrt(xx) )
            if (xi > 0.):
                if (eta > 0.):
                    pass
                else:
                    xx = pi - xx
            else:
                if (eta > 0.):
                    xx = 2.*pi - xx
                else:
                    xx = pi + xx

            # Return the function value
            yy = sqrt( (1.- xmu)*(1.+ xmu) )
            if (m == 1):
                return sqrt(3.)*yy*cos(xx)
            else:
                return sqrt(3.)*yy*sin(xx)

    else:
        print "Error: this version of sph only supports l=0,1"
        exit()


#-----------------------------------------------------------------------
#                       end of sph.py
#-----------------------------------------------------------------------
```